

Настольная книга
пользователя ПК

В. К. Потоцкий

Работаем на языке Си

МП «Малип»

**Настольная книга
пользователя ПК**

В. К. Потоцкий

Работаем на языке Си

**Москва
«Малип»
1992**

ББК 32.97
П64
УДК 681.3

Потоцкий В. К.

**П64 Работаем на языке Си. — М.: МП «Малип», 1992. — 70 с.
(Настольная книга пользователя ПК)**

Описываются средства и технология программирования на языке Си в среде MS-DOS, на персональных ЭВМ типа IBM PC XT/AT. Изложение ориентировано на пользователей систем программирования Turbo-C, MicroSoft C, Quick C. Значительное внимание уделено работе с файлами.

Для широкого круга пользователей ПЭВМ, в том числе желающих получить профессиональную подготовку в области разработки прикладного программного обеспечения.

1. БАЗОВЫЕ СРЕДСТВА ЯЗЫКА

Создание любой программы требует, как правило, выполнения следующих действий: определение местоположения и ввод исходных данных; задание операций над ними; вывод результирующих данных.

Ввод предполагает чтение данных с клавиатуры, диска или порта ввода-вывода.

Определение местоположения данных включает указание типа данных. Данные могут быть представлены в программе значениями (данные-константы) и адресами местоположения значений (данные-переменные).

По структуре различают простые и составные данные.

Простые данные:

а) числа целые (u - unsigned integer, ld - long integer, d - integer) и вещественные (f - floating point, e - floating point in exponential format);

б) символы (с - character).

Составные данные:

а) массивы (данные регулярной структуры);

б) структуры (логически связанные данные) разных типов.

Особое место занимают данные типа "указатель" (ссылка). Значением указателя является адрес простой переменной, структуры или массива. Операции над данными задаются посредством операторов присваивания, передачи управления по условию и организации цикла.

Знаки операций (арифметических, отношения и логических) используются для объединения констант и (или) переменных в соответствующие выражения.

Вывод результирующих данных осуществляется на стандартный терминал (stdout), диск или порт ввода-вывода. Как вывод, так и ввод производится с помощью стандартных программных модулей языка Си, называемых функциями.

1.1. Вывод данных

К основным функциям, реализующим вывод данных, относятся printf(), puts() и putchar().

Обобщенная форма функции printf():

printf(строка форматов, объект, объект, ... , объект);

Пример:

```
printf("Итого %d \n", y);
```

где %d - спецификация поля представления целого числа; y - объект вывода. Объект вывода - это переменная, константа, выражение или указатель функции.

Возможно использование следующих спецификаций полей представления данных: %d - целое число; %u - целое число без знака; %ld - длинное целое число; %e - вещественное число

в форме мантиисы и порядка; %p - значение указателя; %c - символ; %s - строка символов; %x - целое число в шестнадцатеричной форме. Размер поля задается по умолчанию или явно цифрой между знаком "%" и буквой.

Пара символов "\n" называется управляющей последовательностью. Назначение подобных последовательностей: \n - переход на следующую строку; \t - переход к следующей зоне (табуляция); \f - переход к следующей странице; \b - возврат на одну позицию; \\ - вывод знака "\"; \xhhh - вывод символа, ASCII-код которого представлен одной, двумя или тремя шестнадцатеричными символами.

Функции puts() и putchar(), как и функция printf(), служат для вывода информации на экран. Функция puts() используется для вывода строки символа на экран. Строка символов может быть определена в программе в виде константы путем заключения последовательности символов в кавычки.

Например: "Система TURBO C"

Функция puts() выводит строку на экран дисплея, а затем выполняет переход к новой строке поля экрана.

Пример:

```
#include <stdio.h>
main()
{
    puts("Система Turbo C");
}
```

Отметим, что "\n" не используется, так как переход к новой строке происходит автоматически.

Функция putchar() выводит одиночный символ на экран без последующего перехода к новой строке.

Пример:

```
putchar(symb);
```

Может возникнуть сомнение в целесообразности наличия в языке (библиотеке системы программирования) функций puts() и putchar() при наличии функции printf(). Однако printf() является слишком мощной и громоздкой и ее целесообразно применять для числовых данных или при специальном форматировании поля представления данных. Программа с функциями puts() и putchar() выполняется быстрее.

1.2. Строки символов

В языке Си нет типа данных, именуемого "строка символов". Однако фактически имеется возможность определить строку символов и оперировать с ней.

Два способа определения строки символов: в форме массива символов или посредством указателя (ссылки) на символ.

Пример (работа со строкой символов как с массивом символов):

```
#include <stdio.h>
```

```
#include <string.h>
main()
{
char mas[15];
strcpy(mas, "Система Turbo C");
puts(mas);
}
```

Команда

```
char mas[15];
```

“приказывает” компилятору зарезервировать область памяти для 14 символов. 15-я часть общей выделяемой области будет заполнена так называемым нулевым символом, а именно, “\0”.

Переменная `mas` несет в себе информацию об адресе первого из 14 символов, которые будут размещены в резервируемой области. В процессе компиляции по команде

```
strcpy(mas, "Система Turbo C");
```

во-первых, формируется строковая константа “Система Turbo C” с замыкающим ее символом “\0” (ее код - это часть объектного кода программы); во-вторых, создается машинный код вызова функции копирования `strcpy()`.

При выполнении команды

```
puts(mas);
```

значение переменной `mas`, т.е. адрес массива символов (адрес строки символов) передается функции вывода `puts()`. Она проверяет, не является ли символ по адресу `mas` нулевым, т.е. символом “\0”. Если анализируемый символ не является нулевым, то после его вывода на экран функция `puts()` адресуется к следующему символу, расположенному по адресу `(mas + 1)`. Если там нулевой символ, то вывод завершается. Поэтому нет необходимости хранить информацию о длине строки символов. Ее конец всегда отмечен присутствием нулевого символа.

Определение местоположения строки символов и адресация к ней посредством указателя

Переменная типа указатель (ссылка) содержит адрес местоположения объекта, на который эта переменная “ссылается”.

Пример: `char *st;`

Переменная `st` - указатель на символ, т.е. ей должен присваиваться адрес символа.

Пример:

```
#include <stdio.h>
```

```
#include <string.h>
```

```
main()
```

```
{
```

```
char *st;
```

```
st = "Система Turbo C\n";
```

```
puts(st);
```

```
}
```

В результате обработки строки программы

`st = "Система Turbo C\n";`

компилятор, во-первых, создает где-то в тексте объектного кода программы машинный эквивалент строки "Система Turbo C\n"; во-вторых, присваивает переменной `st` адрес первого символа строки, т.е. адрес буквы "С". Машинный код строки завершается кодом нулевого символа (`\0`).

Команда `puts(st);` воспроизводит символы, начиная с того, который адресуется переменной `st`, вплоть до символа-терминатора строки символов (`\0`).

1.3. Операции над данными

Выражение "операции над данными" предполагает наличие объекта (объектов) и использование знака операции. Наиболее часто применяется операция присваивания, знак которой имеет вид `"="`.

Примеры:

`k = 125 ;`

`k = k + 1;`

`r = a = b ;`

Операции с участием одного объекта называются унарными, двух объектов - бинарными. Типовые арифметические операции (`"+"` - сложение, `"-"` - вычитание, `"*"` - умножение, `"/"` - деление) являются бинарными. В языке Си используются унарные операции единичного приращения положительного (`++`) и отрицательного (`--`).

Примеры:

`sum = a + b++;` (апостериорное приращение)

`sum = a + ++b;` (априорное приращение)

Операции над битами (двоичными цифрами) также делятся на бинарные и унарные.

Бинарные операции:

сдвиг влево

`a = b << c;`

сдвиг вправо

`a = b >> c;`

И

`a = b & c;`

ИЛИ

`a = b | c;`

исключающее ИЛИ

`a = b ^ c.`

Унарная операция:

НЕ

`a = ~ b.`

Оператор присвоения

Эта базовая команда (оператор) всех языков программирования высокого уровня имеет следующую обобщенную форму:

`<имя> = <выражение>;`

Разновидностью этой команды является следующая:

`<имя> = <имя> <знак> <выражение>;`

В языке Си последнюю запись можно представить также в виде

`<имя> <знак> = <выражение>;`

В приведенных выше командах “имя” - имя переменной, “знак” - знак операции, “выражение” - совокупность имен переменных и (или) констант, связанных знаками операций.

Примеры:

<code>a = a + b;</code>	<code>a+ = b;</code>
<code>a = a - b;</code>	<code>a- = b;</code>
<code>a = a * b;</code>	<code>a* = b;</code>
<code>a = a % b;</code>	<code>a% = b;</code>
<code>a = a >> b;</code>	<code>a>> = b;</code>
<code>a = a & b;</code>	<code>a& = b;</code>
<code>a = a b;</code>	<code>a = b;</code>
<code>a = a ^ b;</code>	<code>a^ = b;</code>

Помимо арифметических операций и побитовых логических операций в языке Си используются операции отношения и логические операции. Если в результате последних двух операций (отношения и логических) ответ - “ложно”, то значение результата равно нулю. Если же ответом является “истинно”, то значение результата отлично от нуля. В языке Си нет логических (boolean) данных.

Примеры операций отношения:

<code>a > b</code>	<code>a >= b</code>	<code>a < b</code>
<code>a <= b</code>	<code>a == b</code>	<code>a != b</code>

Знаки логических операций:

`a && b` - операция И;
`a || b` - операция ИЛИ;
`! a` - операция НЕ.

Особенность этих операций заключается в прекращении вычисления логического выражения, если результат становится известным еще до выполнения всех действий.

1.4. Операции над адресами

Имеются две основные операции, выполняемые над адресами: определение адреса (&) и обращение по адресу (*). Например, если `sum` - переменная типа `int`, то `&sum` - адрес местоположения `sum`. Если `st` - указатель на местоположение данного типа `char`, то `*st` - данное типа `char`.

Пример программы:

```
#include <stdio.h>
main()
{
    int sum;
    char *st;
    sum = 4 + 5;
    st = "Система Turbo C\n";
    printf("sum = %d &sum = %p\n", sum, &sum);
    printf("*st = %c st = %p\n", *st, st);
}
```

В первой строке выводятся значение переменной `sum` и адрес его (значения) местоположения, во второй строке выводятся символ "C" и его адрес.

1.5. Ввод данных

В большинстве случаев требуются не только обработка данных и их вывод, но и ввод исходной информации. Для ввода в языке Си имеется несколько стандартных функций: одни вводят данные с клавиатуры, другие позволяют брать данные из файла. К их числу прежде всего следует отнести функции `scanf()` и `read()`.

Функция `scanf()`

Используется для интерактивного ввода. Обобщенная форма функции:

`scanf(строка форматов, адрес, адрес, ..., адрес);`

В строке форматов применяются многие из спецификаций, используемых в строке форматов функции `printf()`, а именно: `%d`, `%f`, `%c` и `%s` - описатели полей представления целого числа, вещественного числа, символа и строки символов соответственно.

В отличие от `printf()` функция `scanf()` требует указания в списке вывода не данных, а их адресов.

Пример (ввод двух целых чисел в "ячейки" памяти `a` и `b`):

`scanf("%d%d",&a,&b);`

Знак операции "&" указывает на передачу в функцию `scanf()` адресов ячеек (областей) памяти `a` и `b`.

Пример (ввод строки символов):

```
#include <stdio.h>
```

```
main()
```

```
{  
    char im[12];  
    printf("Ваше имя :");  
    scanf("%s",im);  
    printf("Введено имя : %s \n", im);  
}
```

Здесь `im` - массив символов. Значением `im` является адрес массива. Поэтому перед `im` не используется & - знак операции определения адреса. Отметим, что строка определена путем использования массива символов (описываемого командой `char im[12]`). В данном случае для строки символов резервируется 12 байт памяти.

Другим возможным решением является использование указателя (ссылки) на символ, а именно:

`char *im;`

Однако в таком случае перед вводом строки символов с клавиатуры потребовалось бы выделение памяти (посредством,

например, функции malloc). На начало выделяемой области указывала бы переменная im.

Функции gets() и getch()

Положим, что с клавиатуры нужно вводить строку символов, содержащую пробелы. Например, нужно вводить фамилию и имя. При использовании для ввода функции scanf() как только очередным символом становится пробел, ввод строки символов завершается. (Для scanf() пробел - признак конца вводимой строки.) Поэтому приходится несколько раз обращаться к функции scanf().

Пример:

```
#include <stdio.h>
main()
{
    char fam[20], im[12];
    printf("Ваши фамилия и имя:");
    scanf("%s%s", fam, im);
    printf("Фамилия : %s \t Имя %s \n", fam, im);
}
```

Функция gets() позволяет вводить строку символов, содержащую пробелы.

Пример:

```
#include <stdio.h>
main()
{
    char famim[35];
    printf("Ваши фамилия и имя:");
    gets(famim);
    printf("ФИО: %s \n", famim);
}
```

При использовании функции gets() ввод продолжается до тех пор, пока не будет нажата клавиша "Ввод". Эта функция дополняет строку введенных символов символом "\0" (занимающим первую позицию в таблице ASCII-кодов).

Функция getch() применяется для ввода с клавиатуры одиночного символа без его отображения на экране. Эта функция не требует задания параметра (аргумента) и принимает значение типа char.

Пример (фрагмент программы):

```
...
char ch;
ch = getch();
...
```

1.6. Команды передачи управления

Программа - это описание средствами языка программирования алгоритма решения задачи. Любой алгоритм можно расчленить на

следования, ветвления и циклы. В простейшем случае имеют место только следования без анализа каких-либо условий и принятия решений о дальнейшем ходе вычислений.

Если имеет место проверка условия и продолжение операций по одному из возможных направлений без повторения ранее выполненных действий, то имеет место ветвление. Если же имеет место периодическое повторение действий, то говорят о наличии цикла. Чаще всего ветвление реализуется оператором

if ... else ... ;

Оператор if

Возможно использование этой команды (оператора) в одной из двух форм:

- а) if (выражение)
 оператор-1;
 else
 оператор-2;
- б) if (выражение)
 оператор-1;

Если выражение (некоторое условие) не равно нулю, то выполняется оператор-1. Оператор может быть простым или составным. Составной оператор - это совокупность операторов (команд), заключенных в фигурные скобки { }. Каждый оператор внутри скобок завершается точкой с запятой.

Пример:

```
#include <stdio.h>
main()
{
    float a,b,res;
    printf(" Введите два числа:");
    scanf("%f%f",&a,&b);
    if (b == 0)
        printf("Отношение a/b не определено \n");
    else
    {
        res = a/b;
        printf("a/b равно %f\n",res);
    }
}
```

В приведенном примере функция scanf() позволяет вводить два числа, т.е. имеются два поля ввода.

Общее правило ввода:

а) в очередное поле помещаются все символы вплоть до (но не включая) пробела;

б) вводятся все символы, пока не встретится символ, недопустимый для ввода (например, цифра 9 при вводе восьмеричного числа);

в) воспринимается n символов, если n - ширина поля

представления данного.

Пример:

```
if ((ch = getch()) == 'q')
    puts("Конец работы \n");
else
    puts("Работа продолжена \n");
```

Выражение, используемое после служебного слова if, может включать несколько операций присваивания, разделяемых запятыми и заключаемых в скобки. Значение содержимого скобок определяется последним присваиванием. Это справедливо для любого составного выражения.

Пример:

```
( star = ch, ch = getch() )
```

Операторы организации цикла

Циклические вычислительные процессы реализуются операторами while..., for... или do... while... . Наиболее универсальным является оператор while (выражение) оператор;

Пока выражение не равно нулю, повторяется выполнение действий оператора (простого или составного).

Пример 1:

```
#include <stdio.h>
main()
{
    int dlina;
    dlina = 0;
    puts("Введите строку, затем нажмите клавишу <ВВОД>");
    while ( getchar() != '\n' )
        dlina++;
    printf("\n Длина строки равна %d символам \n",
           dlina);
}
```

Пример 2:

```
#include <stdio.h>
main()
{
    char *text;
    int ind;
    text = "Си - язык профессиональных программистов!";
    ind = 1;
    while (ind <= 5)
    {
        printf("Строка %2d: %s", ind, text);
        ind ++;
    }
}
```

В результате работы программы на экран выводится сообщение вида:

Строка 1: Си - язык профессиональных программистов!

Строка 2: Си - язык профессиональных программистов!

.....

Строка 5: Си - язык профессиональных программистов!

В этой программе реализован "цикл по счетчику". Его можно оформить в более компактной форме:

```
ind=0;
```

```
while ( ind++<=5 )
```

```
    printf("Строка %2d:%S",ind,text);
```

Однако для организации цикла по счетчику язык Си располагает специальным оператором

```
for ( выражение-1; выражение-2; выражение-3 )  
    оператор;
```

где "выражение-1" устанавливает начальное значение параметра цикла, "выражение-2" определяет условие продолжения выполнения операций, "выражение-3" задает правило модификации параметра.

Цикл for можно описать посредством цикла while:

```
    выражение-1;
```

```
    while ( выражение-2 )
```

```
    {
```

```
        оператор;
```

```
        выражение-3;
```

```
    }
```

Отметим, что каждое выражение может быть составлено из нескольких, разделяемых запятыми. Такое выражение называют групповым.

Пример:

```
#include <stdio.h>
```

```
main()
```

```
{
```

```
    char *text;
```

```
    int vverh,vnis;
```

```
    text = "Шар";
```

```
    for (vverh=1,vnis=5;vverh<=5;vverh++,vnis--)
```

```
        printf("%s : %2d \t %2d \n",text,vverh,vnis);
```

```
}
```

Цикл можно организовать посредством оператора

```
do
```

```
    оператор
```

```
    while (выражение);
```

Действия, предписываемые "оператором", продолжаются до тех пор, пока выполняется условие, задаваемое "выражением".

Пример:

```
#include <stdio.h>
```

```
#include <conio.h>
```

```
main()
```

```
{
```

```

float a,b,res;
do {
    puts("\n Введите два числа:");
    scanf ("%f%f",&a,&b);
    if ( b==0 )
        puts("Делить нельзя \n");
    else
    {
        res=a/b;
        printf("Результат : %f\n",res);
    };
    puts("Нажатие клавиши q завершает работу");
} while ( getch() != 'q' );
}

```

Разница между циклами while и do...while в том, что входящий в их состав оператор выполняется обязательно хотя бы один раз в случае использования цикла do...while.

2. ФУНКЦИИ

Если некоторую совокупность операций приходится неоднократно выполнять (часто вводя новые исходные данные), то формируют самостоятельный программный модуль, называемый подпрограммой. В языке Си применяется только одна разновидность подпрограмм - функции.

Согласно определению подпрограммы-функции в результате ее работы формируется некоторое значение, связанное с ее именем. Поэтому при определении подпрограммы перед ее именем ставится указатель типа. Иногда логика использования функции не требует формирования значения, связанного с именем функции. Тогда в качестве указателя типа используется служебное слово void.

В языке Си следует различать определение (объявление) функции и ее описание. Если о наличии функции объявлено, то все остальные функции (включая главную функцию, именуемую main) могут содержать обращения к ней. Отметим, что Си-программа - это совокупность следующих одна за другой функций, одна из которых - main. Однако объявить о функции недостаточно. Необходимо ее описать, т.е. привести ее полный текст.

Пример:

```

#include <stdio.h>
#include <conio.h>

```

```

/* Определение функций */
void inab(float *u1, float *u2);
float result(float chi, float zna);
void outres(float res);

```

```

const float INFINI=3.4E+38;

```

```

/* Текст главного модуля */
main()
{
    float a,b,res;
    do {
        inab(&a,&b); /* Ввод a и b */
        res = result (a,b);
        outres(res); /* Вывод res */
        puts("Для продолжения работы нажмите клавишу Q");
    } while (getch() != 'q' );
} /* Конец модуля main */

/* Описание функции inab */
void inab(float *u1, float *u2)
{
    puts(" Введите два числа:");
    scanf("%f%f", u1, u2);
}

/* Описание функции result */
float result(float chi, float zna)
{
    if ( zna == 0.0 )
        return (INFINI);
    else
        return (chi/zna);
}

/* Описание функции outres */
void outres(float res)
{
    if ( res==INFINI )
        printf("Результат не определен \n");
    else
        printf("Результат :%f \n",res);
}

/* Текст программы завершен */

```

В начале программы дано определение используемых функций. В определении указаны типы и количество параметров, что обеспечивает в дальнейшем при вызове функций контроль возможных ошибок (несоответствие количества и типа фактических параметров). Далее описывается константа. Поскольку описание дано вне всех функций, константа "видна" (доступна) всем функциям. Она используется как индикатор факта деления на нуль. Затем следует текст (описание) главной функции, всегда именуемой main. В программе такая функция - единственная и выполнение всегда начинается с обращения к

функции `main()`. В тексте программы `main()` может располагаться в любом месте. Обычно ей предшествуют только так называемые прототипы функций (объявления о функциях) и описание глобальных переменных и констант. В рассматриваемой программе за `main()` следуют описания трех функций: ввода данных, вычисления результата и его вывода.

Результат выполнения функции `inab()`, связанный с ее именем, объявлен как данное типа `void`, поскольку этот результат не используется (его фактически нет). Параметрами функции являются адреса местоположения подлежащих вводу данных. Поэтому входные параметры определены не как данные типа `float`, а в виде указателей на эти данные.

Функция `result()` получает в качестве исходных данных числовые значения и формирует результат типа `float`. Отметим, что для передачи результата, связанного с именем функции, используется оператор `return`.

Функция `outres()` не возвращает в "точку" вызова результат, связанный с ее именем. Поэтому эта функция имеет тип `void`.

Следует обратить внимание на возможность описания данных как в пределах некоторой функции (подпрограммы), так и вне функций. Константы, типы данных и переменные, определяемые вне какой-либо функции (включая и функцию `main`), являются глобальными по отношению к функции. Это значит, что они могут быть использованы внутри функции.

2.1. Определение функций

Функции, используемые в программе, подлежат определению; говоря иначе, о них должно быть "объявлено". Различают два способа определения функций: классический и современный. Классический способ объявляет только об имени функции и типе возвращаемого в точку вызова результата. Применительно к рассмотренной выше программе имеем

```
void inab();  
float result();  
void outres();
```

Здесь нет никакой информации ни о количестве параметров, ни об их типе.

Современный способ использует объявление, называемое прототипом. Термин "прототип" применяют потому, что объявление о функции включает информацию о параметрах. Прототип содержит информацию, которая потом повторяется при описании функции в ее заголовке. В рассмотренной выше программе прототипы точно повторяют первую строку описания каждой функции. В общем случае прототип подчиняется формату вида

тип имя-функции(инф-парам-1,...,инф-парам-i,...);
где инф-парам-i - информация об i-м параметре (аргументе функции).

В прототипе информация о параметре может быть представлена по форме:

- а) тип
- б) тип имя-параметра
- в) ...

Использование варианта "в" означает, что число параметров может меняться. Применение прототипов позволяет возложить на компилятор ответственность за контроль по выявлению некорректности в задании параметров.

2.2. Описание функций

Программа на языке Си - это совокупность функций. Классический подход к описанию функции подчиняется следующей схеме:

```
тип имя-функции(список имен-параметров)
описание-параметров ;
{
    локальные-описания;
    ...
    команда;
    ...
    команда;
}
```

Современный подход переместил описание параметров внутрь круглых скобок:

```
тип имя-функции(инф-парам-1,...,инф-парам-i,...)
{
    локальные-описания;
    ...
    команда;
    ...
    команда;
}
```

Здесь инф-парам-i - это информация о параметре (имя и тип параметра). Это похоже на прототип: имеется вся информация о каждом параметре. В отличие от прототипа заголовок описания функции не завершается точкой с запятой. Например, при классическом подходе рассмотренная ранее функция ввода двух чисел имела бы вид :

```
void inab(u1, u2)
float *u1, *u2;
{
    ...
}
```

Использованный ранее современный подход обеспечивает описание этой же функции в следующем виде:

```
void inab(float *u1, float *u2)
{
    ...
}
```

Отметим, что описание констант, типов данных и переменных внутри функции (включая и описания внутри main) действуют только в пределах этой функции.

Язык Си не допускает вложения одной функции в другую. (В языке программирования ПАСКАЛЬ, например, это возможно.) Взаимное расположение функций в программе может быть произвольным. Функция, содержащая вызов другой функции, может предшествовать последней. Однако следует избегать обращения к функции, еще не объявленной или не описанной. Встретив такую функцию, компилятор полагает, что она возвращает значение типа int. Если далее в тексте программы тип возвращаемого данного будет определен другим, например char, выдается сообщение об ошибке.

Подводя итоги, можно дать следующую схему структуры программы на языке Си:

- < команды препроцессора >
- < определение типов данных >
- < указание прототипов >
- < переменные >
- < функции >

Структура функции:

```
тип имя-функции(описание параметров)
{
    < локальные описания >
    < команды >
}
```

3. РАСШИРЕНИЯ БАЗОВОГО ПОДМНОЖЕСТВА СРЕДСТВ ЯЗЫКА ПРОГРАММИРОВАНИЯ

3.1. Указатели

Указатель - это переменная, значением которой является адрес оперативной памяти ЭВМ. Память - это набор битов: 8 бит - это группа битов, именуемая байтом; 2 байт образуют слово; 4 байт - длинное слово; 16 байт - параграф. В памяти байты следуют один за другим. Если m - адрес некоторого байта, то $(m-1)$ - адрес предыдущего байта, а $(m+1)$ - последующего. Указатель содержит адрес некоторого байта. Меняя значение указателя, можно "перемещаться" по памяти, адресуясь к различным данным одной и той же или даже разной структуры.

Достоинства применения указателей в следующем:

1) можно помещать данные в память, модифицировать их, извлекать из различных участков памяти (с помощью единственного указателя можно “перемещаться” по совокупности различных данных);

2) в процессе выполнения программы можно создавать новые переменные, т.е. динамически выделять память для данных;

3) задавая приращения значению указателя, можно последовательно адресоваться к различным данным (значение указателя есть сумма адреса сегмента памяти и смещения относительно начала сегмента).

Пример программы:

```
#include <stdio.h>
main()
{
    int x, *u;
    u = &x;
    x = 123;
    printf("адрес x: %p \n", &x);
    printf("значение x: %d \n", x);
    printf("значение u: %p \n", u);
    printf("значение,
           адресуемое переменной u: %d \n", *u);
}
```

Результаты выполнения программы:

адрес x: 168E

значение x: 123

значение u: 168E

значение, адресуемое переменной u: 123

Если в рассмотренной программе оператор

x = 123;

заменить на

*u = 123;

никаких изменений в ее работе на произойдет.

3.2. Динамическое выделение памяти

Для выделения памяти в процессе выполнения программы используется конструкция вида

(тип *) malloc(количество—байт)

где “тип” - тип данных (int, float, char,...)

Пример программы:

```
#include <alloc.h>
main()
{
    int *u;
    u = (int *) malloc(sizeof(int));
    *u = 123;
    printf("значение u: %p \n", u);
    printf("значение,
           на которое указывает u: %d \n", *u);
}
```

Значение функции `sizeof(int)` - это число байтов, оккупируемое данным целого типа (`int`).

На ЭВМ типа IBM PC для целого числа требуется два байта. Функция `malloc(N)` "захватывает" `N` очередных (свободных байтов). Значением функции `malloc()` является адрес первого байта из общего количества "захваченных" ею байтов. Выражение (`int *`) информирует компилятор о том, что адрес первого байта - это указатель на данные конкретного типа, а именно `int`. (Компилятор Turbo C не требует подобной информации, однако этого нельзя сказать о компиляторах языка Си в общем случае.)

Согласно пятой строке программы значение функции `malloc()` присваивается переменной `u`. Это означает, что в данный момент работы программы создается переменная целого типа по имени `*u`. Только после присвоения указателю `u` значения адреса можно присваивать `*u` значение числа, т.е. записывать данное по адресу. Иначе нет уверенности, что для переменной `*u` имеется область памяти, не используемая другими переменными.

3.3. Указатели и функции

В функциях (подпрограммах языка Си) в качестве формальных параметров используются указатели тогда, когда, например, необходимо модифицировать фактические параметры. Говоря иначе, когда в точке вызова необходима модификация данных путем обращения к подпрограмме (функции).

Пример (функция):

```
void swap(int *a, int *b)
{
    int vrem;
    vrem = *a;
    *a = *b;
    *b = vrem;
}
```

В данном случае в описании формальных параметров "сказано", что `a` и `b` - указатели на данные типа `int`, т.е. передаче подлежат адреса (а не значения) переменных целого типа. Выполняемый функцией обмен значений между `*a` и `*b` осуществляется по указанным адресам.

Пример (главная функция, содержащая обращение к функции `swap`):

```
main()
{
    int k,m;
    k = 1991;
    m = 1917;
    printf("До вызова функции k = %5d m = %5d\n",
           k, m);
    swap(&k, &m);
    printf("После вызова k = %5d m = %5d\n",
           k, m);
}
```

3.4. Операции с указателями

Для динамического выделения памяти возможно использование не только функции `malloc()`, но и функции `calloc()`.

Пример программы:

```
#include <stdio.h>
#include <alloc.h>
#define KO 3
main()
{
    int *u, i;
    u = (int *) calloc(KO, sizeof(int));
    *u = 123;
    *(u + 1) = 456;
    *(u + 2) = 1990;
    printf("Адреса переменных:");
    for (i = 0; i < KO; i++)
        printf(" %5p ", (u + i));
    printf("\n Значения переменных:");
    for (i = 0; i < KO; i++)
        printf(" %5d ", * (u + i));
    printf("\n");
}
```

Функция `calloc()` требует задания двух параметров: количество объектов, для которых динамически выделяется память, и размер области памяти в байтах для одного объекта.

Команда `*u = 123;` записывает целое число в область, адрес первого байта которой хранит переменная `u`.

Команда `*(u + 1) = 456;` помещает другое целое число в область памяти, первый байт которой отстоит от начала первой области на длину, равную двум байтам, т.е. размеру области хранения данного типа `int`.

В общем случае $(u + i)$ означает адрес памяти, определяемый выражением

$$u + (i * \text{sizeof}(\text{int}))$$

Результаты работы приведенной выше программы могли бы иметь следующий вид:

Адреса переменных: 02BA 02BC 02BE

Значения переменных: 123 456 1990

Итак, если `u` - указатель на данные некоторого типа, то выражение $(u + i)$ принимает значение, равное

$$u + (i * \text{sizeof}(\text{тип}))$$

3.5. Массивы

Одномерный массив данных описывается оператором вида
тип имя[размер];

Пример программы:

```
#include <stdio.h>
#define KO 3
```

```

main()
{
    int mas[KO], i;
    mas[0] = 123;
    mas[1] = 456;
    mas[2] = 1990;
    printf("Адреса переменных:");
    for (i = 0; i < KO; i++)
        printf("%p", &mas[i]);
    printf("\n Значение переменных:");
    for (i = 0; i < KO; i++)
        printf("%5d", mas[i]);
    printf("\n");
}

```

Элементы массива нумеруются (индексируются) с нуля:

mas[0] mas[1] mas[2]

т.е. последний элемент имеет формат

имя[размер-1]

Количество байтов, резервируемое для массива, определяется выражением

размер * (sizeof(тип))

3.6. Массивы и указатели

Если выполнить последнюю из рассмотренных программ, то получим результат, подобный следующему:

Адреса переменных: 183A 183C 183E

Значения переменных: 123 456 1990

С точностью до значения первого адреса результаты последних программ идентичны.

Отметим, что имя массива можно использовать как указатель. Указатель можно индексировать. При этом справедливо следующее:

(u + i) = &(mas[i])

*(u + i) = mas[i]

Отметим также, что в качестве имени указателя в первой программе (из двух приведенных выше программ) можно использовать имя mas, а во второй программе именем массива может быть u.

В чем же разница между применением указателя и массива для работы с набором однотипных данных? Вводя в программу описание массива, программист "автоматически" резервирует в памяти определенную область. Если же вместо описания массива применяется указатель, то от программиста требуется дополнительно явное указание на выделение памяти (например, функцией malloc) или присвоение указателю адреса ранее выделенной памяти.

3.7. Массивы и строки

Строку символов можно описать, используя понятие массива символов или вводя указатель. Объявляя строку как массив данных типа `char`, мы выделяем для нее память. Используя указатель на данные типа `char` для определения адреса начала строки, мы не требуем выделения памяти до тех пор, пока явно не используем функцию `malloc()` или `calloc()`, либо не присвоим указателю адрес начала уже существующей строки.

3.8. Многомерные массивы

Многомерные массивы описываются согласно следующему формату:

тип имя-массива[размер-1][размер-2]...[размер-N];

Пример (умножения матрицы А размером 3 x 4 на матрицу В размером 4 x 2):

```
#include <stdio.h>
main()
{
    int a[3][4] = {
        { 5, 2, -4, 13 },
        { 4, 9, 0, 6 },
        { 7, -1, 8, 3 }
    };

    int b[4][2] = {
        { 23, -6 },
        { 92, -53 },
        { -6, 0 },
        { 12, 3 }
    };

    int c[3][2], i, j, k;
    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 2; j++)
        {
            c[i][j] = 0;
            for (k = 0; k < 4; k++)
                c[i][j] = a[i][k] * b[k][j];
        }
    };

    for (i = 0; i < 3; i++)
    {
        for (j = 0; j < 2; j++)
        {
            printf ("c[%d][%d] = %d", i, j, c[i][j]);
            printf ("\n");
        }
    }
}
```

Следует обратить внимание на применение формы записи `a[i][k]`, а не принятой во многих языках программирования

$a[i,k]$. Если в языке Си использовать запись $a[i,k]$, то она интерпретируется как $a[k]$, т.е. как вектор-столбец. Это происходит потому, что в составном выражении $[i,k]$ запятая разделяет одно выражение от другого. Согласно правилам языка Си значением всего выражения является результат вычисления последнего, перед которым стоит запятая.

ПРАВИЛО: в языке Си каждый индекс многомерного массива заключается в свои скобки.

Отметим, что у индексированных переменных последующий индекс меняется "быстрее" предыдущего, т.е. в памяти ЭВМ элементы массива $d[3][2]$ располагаются в следующем порядке:

$d[0][0]$ $d[0][1]$ $d[1][0]$ $d[1][1]$ $d[2][0]$ $d[2][1]$

3.9. Массивы и функции

Для передачи в подпрограмму (функцию) в качестве фактического параметра массива достаточно указать адрес его начала.

Пример (функция определения номера минимального элемента одномерного массива целых чисел):

```
int min-indx(int sp[], int ras)
{
    int i, mindx, m;
    mindx = 0;
    m = sp[mindx];
    for (i = 1; i < ras; i++)
        if ( sp[i] < m )
        {
            m = sp[i];
            mindx = i;
        };
    return (mindx);
}
```

Подчеркнем достоинства языка Си: компилятору не требуется размер массива, подлежащего обработке. В тексте функции достаточно указать адрес начала массива.

Пример (программа, содержащая обращение к функции min-indx):

```
#include <stdio.h>
main()
{
    #define dlina 150
    int k, list[dlina];
    for (k = 0; k < dlina; k++)
    {
        list[k] = rand();
    };
    k = min-indx(list, dlina);
    printf(" list[%3d] = %6d \n", k, list[k]);
}
```

Для инициализации одномерного массива случайными целыми

числами можно использовать функцию вида

```
void init(int mas[],int ras)
{
    int k;
    for (k = 0; k < ras; k++)
        mas[k] = rand();
}
```

Если где-то в программе использовать обращение к этой функции вида

```
init(spisok, dlina);
```

то элементы одномерного массива spisok размером dlina получат в качестве значений случайные величины.

Положим, что имеется двумерный массив Table[7][8]. Если, используя функцию init, ввести в программу команду

```
init (Table,7*8);
```

то двумерный массив Table будет воспринят как одномерный, состоящий из $7 * 8 = 56$ элементов. Этим элементам будут присвоены значения датчика случайных чисел.

В функции можно передавать и многомерные массивы.

Пример:

```
void minit(int matrix[][K0], int str)
{
    int i, j;
    for (i = 0; i < str; i++)
    {
        for (j = 0; j < K0; j++)
            matrix[i][j] = rand();
    }
}
```

В рассмотренной функции количество столбцов двумерного массива фиксировано и определяется глобальной переменной K0. Если K0 = 8, то с помощью этой функции можно обработать массив Table, а именно, подав команду

```
minit(Table, 7);
```

Если отказаться от фиксации количества столбцов матрицы, т.е. использовать идентификатор K0 как формальный параметр, то заголовок функции должен иметь вид:

```
void minit(int matrix[][ ],int stro,int K0)
```

3.10. Структуры

Структура - это совокупность взаимосвязанных данных одного типа или разных типов. Взаимосвязь определяет порядок размещения данных.

Пример описания структуры:

```
typedef struct
```

```
{
```

```

char im[10];
char ot[12];
char fa[10];
float oklad;
} sotrudnik;

```

Это описание конкретного вида структуры как типа данных. Здесь `sotrudnik` - идентификатор типа. Далее в программе можно использовать этот тип данных (разновидность структуры) для объявления переменных типа `sotrudnik`.

3.11. Структуры и указатели

Указатели на структуры используются также, как и указатели на данные других типов (`int`, `float`, `char`,...). Если `u` - указатель на структуру, то `(*u)` - имя этой структуры. К конкретному полю естественно адресоваться по схеме

`(*u).имя-поля`

Однако в языке Си принята иная символика для обращения к полю структуры, адресуемой указателем, а именно:

`u->имя-поля`

Пример:

```

#include <stdio.h>
#include <alloc.h>
main()
{
    ...
    sotrudnik *u-st;
    u-st = (sotrudnik *) malloc(sizeof(sotrudnik));
    strcpy(u-st->im, "Иван");
    strcpy(u-st->ot, "Петрович");
    strcpy(u-st->fa, "Орлов");
    u-st->oklad = 540.00;
    ...
}

```

Предполагается, что в начале текста программы описан тип данных `sotrudnik`.

3.12. Передача управления: оператор `switch`

Оператор `switch` обеспечивает выбор возможного "направления" хода вычислений.

Пример (функция, обеспечивающая выбор из некоторого меню):

```

#include <ctype.h>
menu(short *da)
{
    char v;
    *da = 0;
    do {
        v = toupper(getch());
        if (v == 'R') reshnie(da);
    }
}

```

```

        else
            if (v == 'T') test();
            else
                if (v == 'C') clea();
                else
                    if (v == 'E') erro();
                    else
                        vse-dru(v, da);
    } while ( ! *da );
}

```

Эту же задачу решает оператор switch, а именно:

```

#include <ctype.h>
menu(short * da)
{
    char v;
    *da = 0;
    do {
        v = toupper(getch());
        switch (v) {
            case 'R': reshenie(da); break;
            case 'T': test(); break;
            case 'C': clea(); break;
            case 'E': erro(); break;
            default : vse-dru(v, da);
        }
    } while ( ! *da );
}

```

После входа в цикл клавиатура “ожидает” (функция getch) поступление (ввод) символа, который превращается (посредством функции toupper(), описываемой в файле ctype.h) в прописную букву. Выполнение программного модуля продолжается, пока значение переменной *da равно 0.

Оператор выбора (переключения) switch осуществляет проверку выполнения условия (совпадение значения “вводимой” величины со значением “метки”, указываемым после слова case) для каждого случая (case). Если имеется совпадение, то выполняются команды данного “случая” (case). По достижении команды break или конца перебора всех случаев (вариантов) выполнение оператора switch прекращается. Использование фразы default ... необязательно. Если оператор break отсутствует, то осуществляется проверка всех следующих друг за другом случаев (case).

В качестве метки допустимо использование только констант (типа int, char или enum). Объект сравнения с меткой может быть константой, переменной, указателем функции, выражением упомянутого типа.

Если не использовать функцию преобразования в прописную

букву (toupper), то оператор switch принимает вид:

```
switch (v) {  
    case 'r':  
    case 'R': reshenie(da); break;  
    case 't':  
    case 'T': test(); break  
    ...  
}
```

Еще один пример подобного решения (фрагмент программы):
typedef enum {po,vto,sre,che,py,su,vos} den;
main()
{

den sego—dny;
...

```
switch (sego—dny) {  
    case po :  
    case vto:  
    case sre:  
    case che:  
    case py : puts("Трудимся на производстве !");  
               break;  
    case su : puts("Решаем домашние проблемы !");  
    case vos: puts("Отдыхаем !");  
}
```

...
Если значением sego—dny является su, то выводится "Решаем домашние проблемы !", а затем "Отдыхаем !". Правило выполнения оператора switch гласит: если выполняется условие какого-либо случая (case), то выполняются действия, предписываемые всеми последующими случаями (case).

3.13. Передача управления: дополнительные средства ее организации

Операторы передачи управления можно разделить на три группы:

- 1) безусловной передачи управления (goto..., а также вспомогательные операторы return, break, continue);
- 2) передачи управления по условию (if..., if...else..., switch...case...);
- 3) цикла (while..., for..., do...while...).

Оператор goto

Обобщенная форма:

goto метка;

Пример (фрагмент программного модуля):
...

```

    goto mel2;
    ...
mel2: printf("Конец\n");
    ...

```

Имя метки формируется по правилу наименования переменных.

Оператор return

Два варианта применения:

1. Если функция (подпрограмма) возвращает некоторый результат (значение) в точку вызова, то необходимо наличие в тексте функции оператора вида
return (выражение);

Пример:

```

/* Функция, возвращающая значение
   большего из двух целых чисел */
int max(int x, int y)
{
    if ( x > y )
        return (x);
    else
        return (y);
}

```

2. Помимо возврата значения в точку вызова функции оператор return можно применять тогда (в том месте текста функции), когда требуется возврат в вызывающий программный модуль. (Возврат выполняется до завершения всех операций функции.)

Пример:

```

int poisk(int massiv[], int dlina)
{
    int k, indmin, mimas;
    if ( dlina <= 0 )
        return (-1);
    ...
}

```

Если параметр dlina (длина массива) не больше нуля, выполнение программного модуля (функции) прерывается.

Оператор break

Этот оператор (команда) позволяет выйти из цикла прежде, чем цикл завершит свою работу.

Пример:

Постановка задачи: таблица из 100 строк и 10 столбцов заполняется числами, вводимыми с клавиатуры. Если очередное число меньше нуля, заполнение данной строки прекращается. В первую позицию строки записывается количество введенных в "строку" чисел.

```
#include <stdio.h>
```

```

#define LIM 100
#define MAX 10
main()
{
    int i, j, nom;
    int matr[LIM][MAX];
    for ( i = 0; i < LIM; i++ )
    {
        j = 1;
        while ( j < MAX-1 )
        {
            printf("matr[%d][%d] = ", i, j);
            scanf("%d", &nom);
            if ( nom < 0 )
                break;
            matr[i][j++] = nom;
        };
        matr[i][0] = j;
    }
}

```

Ранее оператор break применялся для выхода из оператора switch. Break можно использовать в любом цикле, но не в операторе if...else... . Оператор break нельзя применять с целью выхода из функции.

Оператор continue

Иногда требуется не выход из цикла, а только невыполнение нижеследующих команд и вновь возврат к началу тела цикла. В подобных ситуациях применяется оператор continue.

Пусть в рассмотренном выше примере оператор if... имеет следующий вид:

```

if ( nom < 0 )
    continue;

```

Если условие выполняется, имеет место возврат к условию while, т.е. программа ведет себя так, как будто бы ввод отрицательного числа является ошибкой и требуется иное число. Обратите внимание, что индекс цикла (j) не претерпевает изменения. (Оператор continue возвращает к проверке условия while раньше, чем параметр j получает приращение.)

К средствам передачи управления можно отнести так называемую операцию условия:

выражение-1 ? выражение-2 : выражение-3

Если выражение-1 истинно (т.е. не равно нулю), то результатом операции условия является значение выражения-2. Если выражение-1 ложно (равно нулю), то результат - значение выражения-3.

Пример:

```

/* Функция выбора
   минимального из двух целых чисел */
int min(int a, int b)

```

```
{
    return ( (a<b) ? a : b );
}
```

Заметим, что тело этой функции эквивалентно оператору

```
if ( x < y )
    return (x);
else
    return (y);
```

Вместо подпрограммы (функции) можно использовать макроопределение вида

```
#define min(x,y) ( ( x < y ) ? x : y )
```

Операцию условия можно применять и в других конструкциях, например, операторе присваивания:

```
max = ( a > b ) ? a : b;
```

Подобное выражение называют условным. Оно более компактно, чем оператор if...else... .

3.14. Классы памяти и инициализация переменных

Классы памяти

Класс памяти в языке Си - это понятие, распространяющееся на такие "объекты" языка, как идентификатор переменной и указатель функции. Класс памяти определяет "время жизни" объекта как глобальное или локальное.

Объект с "глобальным временем жизни" (глобальный) существует и имеет значение на протяжении всего выполнения программы. Все функции (их указатели) - глобальны.

Локальные переменные "захватывают" новую область памяти каждый раз, когда управление передается блоку, в котором они описаны. Переменные, описываемые на так называемом внешнем уровне (вне всяких блоков), являются глобальными.

Блок - это последовательность объявлений, описаний и команд, заключенная в фигурные скобки. В языке Си различают два вида блоков: составная команда и описание функции. Описание функции - это некоторая специфическая составная команда с заголовком. Описание функции - это подпрограмма. В нее может быть вложено другое описание функции (другая подпрограмма).

Составная команда может содержать в себе другую составную команду. Глубина вложения, т.е. уровень блока, определяется вычитанием числа правых скобок из числа левых скобок. Самый внешний уровень программы (когда еще нет скобок) равен нулю.

Переменные, описываемые внутри некоторого блока (т.е. на внутреннем уровне) имеют локальное время жизни (локальны). Это значит, что при входе в блок для переменной выделяется некоторая новая область памяти.

Если переменную описать как статическую (static) внутри

блока, то при каждом очередном входе в блок переменная будет помнить свое прежнее место и значение. Это свойство, которое можно охарактеризовать словом “видна”, переменная сохраняет только в пределах собственного блока и блоков, вложенных в него.

Переменная “видна” в пределах файла, функции или блока. Это означает, что к переменной можно обращаться по имени в пределах файла, функции или блока.

Только для меток команд область действия всегда известна заранее: это функция, в которой метка используется. Область действия других объектов определяется уровнем, на котором они описаны (о них объявлено).

```
#include <stdio.h> int k = 1;
main()
{
    printf("%d\n", k);
    /* Выводимое значение k равно 1 */

    { /* Начало 1-го вложенного блока */
        int k = 2, m = 3;
        printf("%d \t %d \n", k, m);
        /* Выводятся числа 2 и 3 */

        { /* Начало 2-го блока */
            int k = 0;
            printf("%d \t %d \n", k, m);
            /* Выводятся числа 0 и 3 */
        } /* Конец 2-го блока */

        printf("%d \n", k);
        /* Выводится число 2 */

    } /* Конец 1-го блока */

    printf("%d\n", k);
    /* Выводится число 1 */

}
```

Пояснение. В этой программе четыре уровня “видности”: внешний уровень и три блока (охватывающий блок - тело функции main и два вложенных один в другой блока).

Объект объявлен на внешнем (external) уровне, если он описан вне всех блоков. Внешний уровень - это уровень файла. Такой объект “виден всем” в файле. Время жизни объекта - глобальное. Это общее правило имеет исключение: на локальном

уровне имя внешнего (глобального) объекта может быть переопределено.

Объект, описанный на внешнем уровне, распространяет область своего действия на весь файл (до его конца). Используя соответствующее описание в другом файле, можно распространить действие этого объекта и на этот (второй) файл.

Если переменная (объект) описана на внешнем уровне с использованием служебного слова `static`, она "видна" только в пределах того файла (с исходным текстом программы), в котором она описана.

Если функция (объект) определена словом `static`, то она тоже "видна" только в пределах файла (программного модуля), где она описана.

Для описания глобальных объектов используются служебные слова `extern` или `static`. Эти слова - описатели класса памяти объекта (переменной или функции). Для локальных объектов (переменных) в качестве описателей используются слова `auto` или `register`. Переменные, описанные внутри блока, действуют в общем случае внутри данного блока и блоков, вложенных в него. Такие переменные - локальные. Действие локальной и даже глобальной переменной может быть отменено в пределах некоторого внутреннего блока, если в нем объявлена переменная с таким же именем.

Различают только два класса памяти (глобальное и локальное время жизни объекта), но используются четыре описателя классов: для глобальных объектов (переменных и функций) - `extern`, `static`; для локальных объектов (переменных) - `extern`, `static`, `auto` или `register`.

Описание переменных на внешнем уровне

Используя описатели `static` и `extern` или опуская их, можно описать переменные как глобальные. Строго говоря, можно описать переменную на внешнем уровне или сослаться на такое описание, сделанное где-либо в другом месте.

При описании имеет место инициализация явно или неявно.

Примеры явной инициализации:

а) `static int k=12;`

б) `int x=7;`

Примеры неявной инициализации:

а) `static int k;`

б) `int x;`

В последнем случае значения `k` и `x` равны нулю.

Описание на внешнем уровне файла (программы) действует только в пределах этого файла, если посредством описателя `extern` не сделать ссылку в другом (втором) файле на описание в первом файле.

Описание на внешнем уровне делается для данной переменной только один раз. Если используется описатель `static`, то одно и то же имя переменной можно использовать в разных файлах. Например, в первом файле, содержащем функцию `main`, может иметь место

```
static int a;
```

а во втором файле, содержащем только функцию `act()`,

```
static int a=2;
```

Описатель класса памяти `extern` позволяет только сослаться на описание, сделанное где-либо в другом месте. Если применить

```
extern int k;
```

то переменной `k` можно воспользоваться до того, как в том же файле она будет описана (например, `int k=3;`) или если она определена в другом файле.

Команда

```
extern тип...;
```

не может содержать знак `"="` как часть инициализации переменной, поскольку `extern` означает, что переменная инициализируется (ее значение описывается) где-то в другом месте.

Для того, чтобы команда `extern...` корректно выполнялась, описание переменной на внешнем уровне должно существовать один и только один раз в любом из файлов, совокупность которых образуют текст программы.

Если определение глобальной переменной содержится в тексте программы более одного раза (например, два раза), то первое определение (описание) трактуется системой как "ссылка" на последующее описание, т.е. как команда `extern`.

Описание класса памяти посредством служебного слова `register` предполагает хранение значения переменной в регистре процессора.

Это ускоряет вычисления. Регистровая переменная "видна" также, как и автоматическая. (В пределах блока, где она описана.) Если нет свободного регистра, то переменная определяется как автоматическая.

Операция получения адреса переменной (&имя-переменной) к регистровой переменной не применима. Регистровая переменная имеет тип `int` (с гарантией). Если размер адреса "вписывается" в поле `int`, то регистровая переменная может быть указателем. По умолчанию ни регистровая, ни автоматическая переменные не получают начальных значений.

Переменная, описываемая на внутреннем уровне как имеющая класс памяти `static`, имеет глобальное время жизни. Однако она "видима" только в пределах того блока, где описана. В отличие от автоматической переменной, статическая сохраняет свое значение после выхода из блока. Статической переменной можно

присвоить значение с помощью константного выражения. По умолчанию ей присваивается первоначально нулевое значение.

Ниже приводятся тексты двух программ, оперирующих переменными различных классов памяти.

Пример 1

Файл 1

```
#include <stdio.h>
extern int k;
main()
{
    k++;
    printf("%d \n", k);    /* k = 4 */
}
int k=3;

next()
{
    k++;
    printf("%d \n", k);    /* k = 5 */
}
```

Файл 2

```
extern int k;
other()
{
    k++;
    printf("%d \n", k);    /* k = 6 */
}
```

Пример 2

```
#include <stdio.h>
int i = 1;
main()
{
    extern int i;
    static int a;
    register int b = 0;
    int c = 0;
    printf("%d \n %d \n %d \n", i, a, b, c);
    /* i=1  a=0  b=0  c=0 */

    act();
}
```

```

act()
{
    static int *ext-i = &i;
    int i = 16;
    static int a = 2;
    a += 2;
    printf("%d \n %d \n %d \n", i, a, *ext-i);
    /* i=16  a=4  *ext-i=1 */
}

```

Инициализация статической переменной имеет место только один раз. При повторном входе в блок повторная инициализация не выполняется.

Класс памяти переменной на внутреннем уровне может быть объявлен посредством описателя

extern тип имя;

Этот описатель говорит, что переменная с указанным “именем” где-то определена на внешнем уровне в одном из файлов, совокупность которых образует исходный текст программы.

Описание типа

extern тип имя;

делает переменную “видимой” в пределах данного блока.

Описание функции на внешнем и внутреннем уровнях

Функции всегда имеют глобальное время жизни. Для них класс памяти может быть определен с помощью описателей static или extern. Если функция определена как static, она “видна” только в том файле, где описана. Функции, находящиеся со “статической” функцией в одном и том же файле, могут ее вызывать. Однако функциям другого файла (с исходным текстом) это не позволено. В другом файле может быть другая статическая функция с этим же именем.

Функции, описываемые с помощью служебного слова extern, “видны” во всех файлах, совокупность которых есть исходный текст программы. Любая функция может вызвать функцию, описываемую с помощью слова extern. По умолчанию (если явно не указан класс памяти) любая функция считается функцией типа extern.

Инициализация переменных базового типа

Инициализация (явная) предполагает при описании указание после имени переменной конструкции вида

= <инициализатор>

При этом действуют следующие правила инициализации:

1) Если описание начинается символом `extern`, явная инициализация недопустима.

2) Переменные, описываемые на внешнем уровне, можно явно инициализировать. По умолчанию они получают значение 0.

3) Если переменная описывается как статическая (`static`), ей можно присвоить значение константного выражения. По умолчанию она получает нулевое значение.

4) Автоматические (`auto`) и регистровые (`register`) переменные инициализируются при каждом входе в блок, где они описаны. Если инициализатор явно не указан, значение автоматической или регистровой переменной не определено.

5) Данные составного типа (массивы, структуры, множества) можно инициализировать только, если для них выбран класс памяти статический (`static`).

6) Для инициализации переменных на внешнем уровне, а также для всех статических переменных как на внешнем, так и на внутреннем уровнях должны использоваться константные выражения.

Примеры:

a) `int X = 12;` Переменная `X` инициализируется константным выражением.

b) `register int *PX = 0;` Ссылка (указатель) `PX` получает значение 0; создается ссылка в никуда (ссылка типа `NULL`).

c) `const int C = (5*1024);` Переменная `C` инициализируется значением, которое затем нельзя изменить (`C` - "постоянная переменная").

d) `int *W = &Y;`

`int *const Z = &S;` Указатель `W` инициализируется адресом некоторой переменной `Y`. Указатель `Z` инициализируется адресом переменной `S`. После этого значение `Z` изменить нельзя. `Z` - ссылка на одну и ту же область памяти.

e) `int GLOB;`

`int func(void)`

{

`int LOC;`

`static int *L-P = &LOC;`

`static int *G-P = &GLOB;`

`register int *R-P = &LOC;`

}

В последнем примере переменная `GLOB` описана на внешнем уровне. Ее "время жизни" - глобальное. Переменная `LOG` описана на внутреннем уровне. Она является автоматической переменной (класс памяти `auto`). Ее адрес определяется (устанавливается) только в момент выполнения функции `func`. Следовательно попытка инициализировать этим адресом статическую адресную переменную `L-P` некорректна. В то же время статическая адресная переменная инициализируется адресом переменной `GLOB`. Ее адрес всегда один и тот же. (Вторая инициализация корректна.) И, наконец, поскольку `R-P` - локальная переменная (регистровая), она может использовать "инициализатор", не являющийся константой.

Инициализация переменных составного типа

В общем случае схема инициализации имеет вид:

= { <список-инициализаторов> }

Объекты, составляющие список, разделяются запятыми. Каждый объект - это константное выражение (одиночный инициализатор) или список инициализаторов. Последний в свою очередь заключается в фигурные скобки. Если список содержит меньше объектов, чем того требует описание составного типа, то оставшиеся элементы (например, поля структуры) инициализируются нулями. Если же в списке объектов больше, чем требуется, регистрируется ошибка.

Пример:

```
int p[4][3] = {  
    {1,1,1},  
    {2,2,2},  
    {3,3,3},  
    {4,4,4},  
};
```

Такая конструкция эквивалентна следующей:

```
int p[4][3] = {1,1,1,2,2,2,3,3,3,4,4,4};
```

Если элементом составного данного является составное данное, внутренние скобки играют важную роль.

Пример:

```
typedef struct {  
    int m1,m2,m3;  
} triplet;  
  
triplet mlist[2,3] = {  
    {{1,2,3},{4,5,6},{7,8,9}},  
    {{10,11,12},{13,14,15},{16,17,18}}  
};
```

Это массив структур. Размер массива равен 2×3 . Каждый элемент массива - это структура из трех чисел. Элемент массива `mlist[0][0]` содержит 1,2,3; элемент `mlist[0][1]` объединяет 4,5,6 и т.д. Первая строка - это массив `mlist[0]` из трех структур. Внешние скобки в этой строке ограничивают область описания этого массива. Если внешние скобки первой строки убрать, произойдет ошибка, например:

```
triplet mlist[2][3] = {  
    {1,2,3},{4,5,6},{7,8,9},  
    {10,11,12},{13,14,15},{16,17,18}  
};
```

В этом случае значения 1,2,3 получит первая структура. Когда компилятор "обнаружит" правую закрывающую скобку (после числа 3), инициализация `mlist[0]` закончится и две остальные структуры двумерного массива получат нулевые значения для элементов. Аналогично будут инициализированы структуры второй строки, т.е. массива `mlist[1]`. Он "использует" числа 4,5,6.

Компилятор попытается “присвоить” числа 7,8,9 массиву `mlist[2]`. Однако исходный массив имеет только две строки и массива `mlist[2]` нет. Регистрируется ошибка. Ее фиксирует компилятор.

Еще один пример:

```
struct R {
    int i,j,k;
    float mass[2][3];
} X = {
    1,
    2,
    3,
    {4.0,4.0,4.0}
};
```

Здесь первые три элемента структуры `R` по имени `X` получают значение 1,2 и 3. Затем первая строка массива `mass` получит значение 4.0 для всех элементов. Остальные элементы `mass` станут нулевыми.

Объединения и их инициализация

К данным составного типа помимо массивов и структур относятся объединения (`union`). Объединение (смесь) описывает переменную, которая может иметь любой тип из некоторого множества типов.

Формат описания объединения:

```
union имя-объединения {
    описания-элементов
} имя-переменной;
```

Возможно альтернативное решение:

```
typedef union {
    описания-элементов
} имя-объединения;
имя-объединения имя-переменной;
```

Пример:

```
typedef union {
    long p-long;
    char *p-ch[4];
} p-word;
```

Данные типа “объединение” занимают область памяти, необходимую для размещения большего из своих элементов.

Переменные типа `p-word` описываются командами вида

```
p-word x;
p-word *u;
```

Альтернативная форма описания:

```
union p-word {
    long p-long;
    char *p-ch[4];
} x, *u;
```

Объединение инициализируется всегда только одним константным выражением, значение которого присваивается первому из элементов объединения (смеси).

Пример:

```
union {  
    char x[2][3]  
    int i,j,k;  
} F = {{  
    {'1'},  
    {'4'}}  
};
```

Здесь переменная F, тип которой - указанное объединение, получает начальные значения. Первый элемент объединения - массив. Поэтому инициализатор составной. Список {'1'} присваивает значения первой строке массива. Первый элемент в строке получает значение '1', а для двух других элементов ничего нет. Они получают нулевые значения. Аналогично "обрабатывается" вторая строка массива: значение '4' получает первый ее элемент и нулевые значения два других элемента.

Инициализация массива символов

Строка символов - это массив одиночных символов. Инициализация строки символов - это инициализация массива.

Пример 1:

```
char c[] = "abc";
```

В результате имеем массив из четырех символов. Четвертый элемент - "терминатор" строки символов (null или \0).

Пример 2:

```
char [3] = "abcd";
```

Используются только первые три символа (a,b,c). Символ 'd' отбрасывается. Нет места и для терминатора строки. В результате формируется "незавершенная" строка символов (она же массив символов), так как нет маркера конца строки. Диагностическое сообщение об ошибке фиксирует эту ситуацию. Если же строка-инициализатор "короче" инициализируемого массива, "оставшиеся" элементы массива получают нулевые значения.

4. ПРЕПРОЦЕССОР И ЕГО ДИРЕКТИВЫ В ПРОГРАММЕ

Это текстовый процессор, манипулирующий исходным текстом Си-программы на первой фазе ее компиляции. Компилятор вызывает препроцессор во время своего первого "прохода". (В принципе программу можно обработать препроцессором и без компиляции.) Если первым символом строки программы является "решетка" (#), то эта строка - директива препроцессора. Использование директив в программе облегчает ее модификацию

(если это необходимо), а также упрощает компиляцию в различных условиях (средах).

Директивы “приказывают” препроцессору заменить тот или иной идентификатор некоторым значением либо выполнить операции макроопределения (директива `#define`), включить текст некоторого файла в данный файл, т.е. программу (директива `#include`), “подавить” компиляцию части текста программы посредством так называемой условной компиляции (директива `#if...`), изменить номер текущей строки и имя компилируемого файла (директива `#line` константа имя).

4.1. Замена идентификаторов

Эта операция реализуется директивой `#define`. Например,

```
#define MAX 10
```

заменяет каждое вхождение идентификатора `MAX` в тексте программы на целое число `10`;

```
#undef MAX
```

отменяет предыдущее определение `MAX`.

4.2. Макроопределения

Для описания макроса (его “макроопределения”) используется также директива `#define`.

Например:

```
#define min(x,y) ( (x < y) ? x : y )
```

Каждое вхождение выражения

```
min(аргумент-1, аргумент-2)
```

в тексте программы заменяется на

```
((аргумент-1 < аргумент-2) ? аргумент-1 : аргумент-2)
```

Так макрос `min(a,b)` заменяется на `( (a < b) ? a : b )`

4.3. Включение файлов

Включение в текст программного модуля некоторого фрагмента на языке программирования реализуется директивой `#include`:

```
#include <имя-файла>
```

или

```
#include "имя-файла"
```

Препроцессор заменяет такую строку содержимым файла с указанным именем. Если имя заключено в скобки, то файл берется из некоторого “стандартного” каталога (например, `tc\include`), а текущий каталог не просматривается. Если имя файла дано в кавычках, то препроцессор ищет файл в текущем каталоге. Если его там нет, то поиск ведется по установленному маршруту и только затем в стандартном каталоге.

Пример:

```
#include <stdio.h>
main()
{
    ...
}
```

К тексту данной программы присоединяется текст файла `stdio.h`.

Система программирования на языке Си (и в частности, например, Turbo C) оснащена библиотеками функций и макроопределений (макросов). Для Turbo C версии 2.0 их более 450. Эти функции и макросы можно вызывать из Си-программы для выполнения процедур ввода-вывода, манипуляций со строками символов и файлами данных, выделения областей памяти во время работы программы, конвертирования данных, выполнения арифметических операций и т.п.

Прототипы этих библиотечных функций определены (объявлены) в том или ином файле-заголовке (header file), имеющем расширение имени `“.h“`. Каждый файл-заголовок, т.е. файл типа `“имя.h“`, включает прототипы соответствующих функций, а также описания глобальных переменных языка Си и библиотечных функций.

На персональных ЭВМ типа IBM PC XT/AT язык Си получил наибольшее распространение в версиях MicroSoft C и Turbo C. Следует также принимать во внимание, что существует стандарт языка Си, известный под именем ANSI C. Между этими тремя главными версиями языка Си имеются некоторые различия.

В системе программирования Turbo C версии 2.0 широко используются файлы-заголовки (имя.h) следующих наименований:

- `alloc.h` - выделение памяти;
- `graphics.h` - описание прототипов графических функций;
- `stdio.h` - описание типов и макросов для стандартных процедур ввода-вывода, а также стандартных “поток” `stdin, stdout, stdprn` и `stderr`;
- `string.h` - определение нескольких функций манипуляций со строками символов и областями памяти;
- `dos.h` - описание констант и ряд определений для вызова утилит DOS;
- `mem.h` - определение нескольких функций манипуляций с областями памяти (в дополнение к `string.h`) и т.д. (всего 29 наименований).

Базовый набор файлов-заголовков MicroSoft C подобен аналогичному набору Turbo C.

Отметим еще раз, что файлы-заголовки содержат только объявления о библиотечных функциях (в них дано определение прототипов функций). Сами функции содержатся в соответствующих библиотеках. (Для Turbo C - это библиотеки `Sx.LIB` `MATHx.LIB` и `GRAPHICS.LIB`.) Символ `“x“` в наименовании библиотеки подлежит замене на букву,

- обозначающую одну из моделей памяти (всего 6 моделей).

Пояснение.

В системах программирования на языке Си принято говорить о различных моделях использования памяти программой пользователя. Применительно к Turbo C различают шесть моделей (tiny, small, medium, compact, large, huge). Они отличаются диапазоном доступа (для программы и данных) к различным частям памяти (начиная от сегмента в 64 Кбайт и вплоть до 1 Мбайта).

4.4. Компиляция по условию

Так называемая условная компиляция осуществляется посредством директивы `#if...`, имеющей следующий обобщенный формат:

```
if выражение—константа—1
    [ блок текста ]
    [ #elif выражение—константа—2
        блок текста ]
    [ #elif выражение—константа—3
        блок текста ]
    ...
    [ #else
        блок текста ]
#endif
```

Если “выражение—константа” не равно нулю, то следующий блок текста подлежит компиляции. Если все выражения—константы равны нулю, компилируется блок текста, указанный после фразы `#else...`. Заключаемое в прямоугольные скобки может отсутствовать.

Выражение—константа - это выражение из различных констант, исключая константы типа `enum` (перечисляемые); недопустимо использование выражения вида `sizeof()`. Однако можно применять выражения вида

`defined(идентификатор).`

Такое выражение не равно нулю (истинно), если идентификатор (имя) ранее определен.

Пример:

```
#if defined(CREDIT)
    credit();
    #elif defined(DEBIT)
        debit();
    #else
        printerror();
#endif
```

Если идентификатор `CREDIT` определен, то компилируется обращение к функции `credit()`. Если идентификатор `DEBIT`

определен, то компилируется обращение к функции `debit()`.
Иначе компилируется вызов функции `printererror()`.

Замечание 1. Согласно языку Си `CREDIT` и `credit` - различные идентификаторы.

Замечание 2. Вместо `#if defined(..)` можно применять `ifndef..`, а именно:

`#ifndef` идентификатор

...

Здесь компиляция имеет место, если идентификатор определен.
Возможна альтернативная конструкция:

`#ifndef` идентификатор

...

обеспечивающая компиляцию, если идентификатор не определен.

Достоинством конструкции

`#if defined(идентификатор)`

...

является возможность построения комплексных выражений вида

`#if defined(A) || defined(B)`

Отметим также, что Turbo C допускает использование `sizeof`.

Пример:

```
#if ( sizeof(void *) == 2 )
```

```
    #define RDATA
```

```
    #else
```

```
    #define LDATA
```

```
#endif
```

Директива `#define RDATA` означает, что препроцессор удаляет из текста (делает пустым) идентификатор `RDATA`. Если бы использовалась конструкция вида

`#define RDATA 125`

то каждый раз вместо `RDATA` препроцессор размещал бы 125.

Директива

`#line` константа ["имя"]

позволяет, задав константу в виде целого числа, определить номер строки программы. (Вместо номера, устанавливаемого для файла компилятором.) При отладке программист "узнает" этот номер. Задав "имя" (в кавычках), можно изменить имя компилируемого файла (программы).

5. ФАЙЛЫ ДАННЫХ И ФУНКЦИИ ВВОДА-ВЫВОДА

Функции (подпрограммы) ввода-вывода, хранимые в стандартной библиотеке Си, обеспечивают чтение данных из файла или внешнего устройства (интерпретируемого как файл), а также запись данных в файл или на внешнее устройство (например, экран дисплея).

Структура файла данных в языке Си никак специально не

предопределена. В общем случае поток данных - это последовательность байтов.

Все библиотечные функции ввода-вывода можно разделить на три группы: функции, оперирующие с "потокom"; файлами на "нижнем уровне"; консолью и портами ввода-вывода.

5.1. Функции, оперирующие с потоком

Поток - это файл данных или физическое устройство (принтер, дисплей и т.п.), которым программист манипулирует с помощью указателя на объект, именуемый FILE. Этот объект описан в файле `stdio.h`. Фактически это описание структуры (тип которой имеет имя FILE). Такая структура предназначена для хранения информации о файле. Подобная информация включает данные о текущей позиции потока, данные об указателях на используемые буферы, значения индикаторов конца файла и возникновения ошибок при выполнении тех или иных операций.

Функция, открывающая поток данных, возвращает в качестве значения, связанного с ее именем, значение указателя на объект (структуру) типа FILE. Для каждого открываемого потока формируется своя структура типа FILE. Такие структуры образуют массив, который является "производной" файла `stdio.h`.

Следует помнить, что программа пользователя никогда не должна пытаться формировать объект типа FILE. Не следует копировать его. Пользователь в программе только описывает некоторую переменную как указатель на объект типа FILE, а затем присваивает ей значение, возвращаемое функцией открытия файла как потока.

Пример (фрагмент программы):

```
...  
FILE *f-ukazat;  
f-ukazat = fopen("test.doc", "w");  
...
```

В результате открытия файла как потока устанавливается связь между потоком (его представляет указатель `f-ukazat`) и файлом на диске (например, `test.doc`). При этом задается режим доступа к файлу:

- r - чтение существующего файла;
- w - запись в файл;
- a - добавление в конец файла (если файла нет, он создается);
- r+ - чтение и запись (файл должен существовать);
- w+ - чтение и запись в файл (если файл существует, то его содержимое уничтожается);
- a+ - чтение и добавление в файл.

К этим спецификаторам режима доступа может быть присоединен указатель типа потока (`t` или `b`). Так, например,

эквивалентными являются следующие спецификации:

r+b

и

gb+

Различают так называемые текстовые (t) и двоичные (b) потоки. Можно провести определенную аналогию между текстовым потоком в языке Си и текстовым файлом в среде операционной системы MS-DOS. Текстовый файл создается текстовым редактором как строки символов. В конце очередной строки текста помещается два символа CR (возврат каретки) и LF (перевод строки).

Система программирования на языке Си определяет текстовый поток аналогично за одним исключением. Строки текста в потоке отделяются одна от другой не двумя специальными символами, а одним - LF. Поэтому при вводе текстового файла как текстового потока система Turbo C заменяет (преобразует) пару CR/LF в LF. При выводе текстового потока в среду операционной системы как текстового файла одиночный символ LF преобразуется в пару CR/LF.

Двоичный (бинарный) поток по своей организации значительно проще: никаких подобных преобразований система не выполняет. Каждый символ просто вводится или выводится.

Отметим, что система программирования на языке Си “не помнит”, каким потоком был создан данный файл или как читался он последний раз (как текстовый поток или как двоичный). Если при открытии файла не указан тип потока (текстовый или двоичный), то по умолчанию файл обрабатывается системой как текстовый поток. (Об этом свидетельствует специальный индикатор - глобальная системная переменная - fmode.)

Итак, открыть файл как поток можно посредством библиотечной функции

foren(имя— файла, режим— доступа)

Прототип функции foren() помещают в начале программы, указав команду препроцессора

#include <stdio.h>

В файле stdio.h системы Turbo C прототип функции foren() находится в виде

FILE *foren (const char * имя—файла,
const char *имя—доступа);

Файл stdio.h системы MicroSoft C содержит объявление (определение) функции foren в несколько иной форме:

FILE *foren (имя—файла, режим—доступа);
const char *имя—файла;
const char *режим—доступа;

В результате выполнения функции foren() с ее именем оказывается “связанным” некоторое значение, которое является указателем на объект (структуру) типа FILE. Если в процессе

выполнения функции `fopen()` обнаруживается ошибка, значение указателя равно `NULL`.

Если файл успешно открыт как поток, то его указатель можно использовать в различных функциях, оперирующих с потоком. Таких функций насчитывается более 40.

Пример (программа создания копии некоторого файла):

```
#include <stdio.h>
main()
{
    FILE *or , *co;
    if (( or = fopen("TO1.DOC","rt")) == NULL)
    {
        printf("Открыть файл-источник не удастся\n");
        return;
    };
    if (( co = fopen("TO1.BAK", "wt")) == NULL)
    {
        printf("Открыть файл-приемник не удастся\n");
        return;
    };
    while ( !feof(or) )
        fputc(fgetc(or),co);
    fclose (or);
    fclose (co);
}
```

Здесь `feof()` является библиотечным макросом, прототип которого имеет вид:

`int feof(FILE *u);`

Этот прототип определен в файле `stdio.h`.

После первой попытки процедуры чтения файла выйти за его пределы `feof()` становится ненулевой величиной. Если текущая позиция в файле не является его концом, значение `feof()` равно нулю.

Функция

`fputc(символ,поток)`

помещает символ в поток. В файле `stdio.h` ее прототип имеет вид:

`int fputc(int c, FILE *p);`

При успешном выполнении эта функция принимает значение кода символа; при возникновении ошибки - значение EOF. (EOF - индикатор конца файла.) Физически в языке Си конец файла маркируется символом `Ctrl+Z`. EOF принимает значение целого числа.

Функция

`fgets(поток)`

берет символ из потока. В файле `stdio.h` ее прототип имеет вид:

`int fgets(FILE *p);`

При успешном выполнении эта функция принимает значение "прочитанного" символа, преобразовав его в целое число без

знака. Если же эта функция “встречает” конец файла или в процессе выполнения возникает ошибка, функция `fgets()` получает значение EOF.

Функция

`fclose(поток)`

закрывает поток. При успешном завершении своих действий значением `fclose()` является нуль, иначе - EOF.

Для одновременного закрытия всех потоков используют функцию

`fcloseall()`

Ее прототип:

`int fcloseall(void);`

5.2. Потоки, открываемые системой по умолчанию

Когда программа начинает выполняться, автоматически открываются пять потоков, определенных в системе:

`stdin` - стандартный ввод;

`stdout` - стандартный вывод;

`stderr` - стандартный приемник сообщений об ошибках;

`stdaux` - стандартное внешнее устройство;

`stdprn` - стандартный принтер.

По умолчанию роль физического устройства для первых трех потоков выполняет консоль. На ее экран (дисплей) библиотечные процедуры выявления ошибок посылают диагностические сообщения. Поток `stdaux` обычно указывает порт связи с каким-либо внешним устройством. Физические устройства для `stdaux` и `stdprn` определяет конкретная конфигурация средств вычислительной системы.

Названные имена (указатели стандартных потоков) можно использовать в функциях, оперирующих с потоками. Некоторые функции предопределены для работы со стандартными потоками. Например, функция `getchar()` читает символ из потока `stdin`; а функция `putchar(символ)` помещает символ в поток `stdout`.

Стандартные потоки `stdin`, `stdout`, `stderr`, `stdaux`, `stdprn` по умолчанию и (или) явно назначенные определенным внешним устройствам, можно переопределить. Например, назначить для них диск и т.п. Для этой цели можно воспользоваться средствами MS-DOS или библиотечной функцией языка Си `freopen()`.

В файле `stdio.h` эта функция имеет прототип

`FILE *freopen(const char *имя-файла,
const char *режим-доступа, FILE *поток);`

Функция `freopen()` “помещает” имя файла на место потока.

Поток закрывается. Функцию `freopen()` применяют для замены файлов, назначенных потоком `stdin`, `stdout`, `stderr`. Параметр “режим—доступа” подобен одноименному параметру функции `fopen()`.

Пример:

/* В данной программе стандартное назначение консоли для потока `stdout` заменяется назначением этому потоку файла на диске */

```
#include <stdio.h>
#include <process.h>
FILE *po;
main()
{
    /* Назначение для stdout файла по имени KLM */
    po = freopen("KLM","w",stdout);
    if (po == NULL)
        printf("Ошибка при выполнении freopen\n");
    else
    {
        fprintf(po,"Успешное выполнение freopen\n");
        system("type KLM");
    }
}
```

Функция `fprintf()` помещает форматированные данные в поток. По форме эта функция подобна функции `printf()`, которая помещает данные в стандартный поток `stdout`. Если выполнение `fprintf()` успешно, то с ее именем связано количество переданных в поток байтов. При обнаружении ошибки эта функция в качестве значения возвращает EOF. В файле `stdio.h` эта функция имеет прототип:

```
int fprintf(FILE *поток, const char *формат,
            аргумент,...,аргумент,...);
```

Следовательно, поток может содержать форматированные данные. Аналогично функции `scanf()`, читающей форматированные данные из стандартного потока `stdin`, для чтения форматированных данных потока используется функция `fscanf()`.

Функция `system()` позволяет выполнить команду DOS и имеет в файле `process.h` прототип

```
int system(const char *команда);
```

Эта функция позволяет выполнить одиночную команду MS-DOS, пакет команд (batch-файл) или другую программу (файл типа EXE или COM). С именем функции `system()` связан целочисленный результат: 0 - успешное выполнение функции; 1 - неудача.

Пример:

/* Программа записи и чтения форматированных данных файла */

```
#include <stdio.h>
FILE *uk;
long lo;
float fp;
char st[20], ch;
```

```

main()
{
    uk = fopen("test.doc","w+");

    /* Запись данных в файл */
    fprintf ( uk," %s %ld %f %c %c",
        "Данные: ", 56000 , 3.14159 , 'y', EOF );

    /* Указатель файла устанавливается на начало файла */
    fseek(uk, 0, SEEK-SET);

    /* Чтение данных файла */
    fscanf (uk,"%s%l%f%c", st, &lo, &fp, &ch);

    /* Вывод прочитанных из файла данных на консоль */
    printf("%s\n", st);
    printf("%ld\n", lo);
    printf("%f\n", fp);
    printf ("%c\n", ch);
}

```

Функция `fseek()` меняет местоположение указателя текущего положения в открытом файле (потоке). В файле `stdio.h` ей соответствует прототип вида

```
int fseek(FILE *поток, long int смещение, int откуда);
```

Эта функция перемещает указатель потока на задаваемое "смещением" число байтов, начиная с позиции, определяемой параметром "откуда". Параметр "откуда" должен быть одной из констант, определяемой в `stdio.h`, а именно:

`SEEK-SET` - начало файла;

`SEEK-CUR` - текущая позиция указателя файла;

`SEEK-END` - конец файла.

Эта функция позволяет переместить указатель за конец файла. Однако его (указатель) нельзя разместить перед началом файла. Успешно завершив операции, `fseek()` "возвращает" в качестве значения 0. В случае неуспеха это значение отлично от нуля.

Для определения текущей позиции указателя файла, открытого как поток, используется функция `ftell()`. Ее прототип в файле `stdio.h` имеет вид:

```
long int ftell(FILE *str);
```

где `str` - указатель потока. Эта функция принимает значение смещения в байтах относительно начала файла. При неудачном выполнении возвращаемым значением является `-1L` ("длинная отрицательная единица").

Пример:

```

/* Программа, открывающая файл как двоичный (бинарный) поток, читающая
100 символов, а затем определяющая текущую позицию указателя файла */
#include <stdio.h>
FILE *gt;
long pos;
char list[100];

```

```

main()
{
    gt = fopen("test","rb");

    /* Чтение данных */
    fread (list,sizeof(char),100,gt);

    /* Определение позиции указателя файла */
    pos = ftell(gt);
    printf(" Позиция : %ld \n", pos);
}

```

Функция `fread()` читает данные из потока. В файле `stdio.h` эта функция имеет прототип вида:

тип `fread(void *указатель, тип размер,
тип количество, FILE *поток);`

Функция `fread()` читает из потока указанное "количество" данных. Каждое данное имеет длину (в байтах), определяемую параметром "размер". Читаемые данные помещаются в область, адресуемую "указателем".

Функции `fread()` "соответствует" функция `fwrite()`. Обе функции имеют одинаковый перечень аргументов, а именно:

`fwrite(указатель, размер, количество, поток);`

Функция `fwrite()` добавляет в файл, открытый как "поток", указанное "количество" данных заданного "размера". Местоположение данных определяет "указатель".

5.3. Буферизация файла, открываемого как поток

Открываемому как поток файлу назначается по умолчанию буфер. Размер буфера определяет константа `BUFSIZ`, определяемая в файле `stdio.h`. Указанное справедливо для всех файлов, кроме стандартных `stdin`, `stdout`, `stderr`, `stdaux`, `stdprn`. Потоки `stdaux` и `stderr` не буферизуются по умолчанию. Исключением являются случаи их применения в функциях типа `printf()` и `scanf()`. В таких случаях им назначается временный буфер. Эти два потока (`stdaux` и `stderr`) можно буферизовать посредством функций `setbuf()` и `setvbuf()`.

Потоки `stdin`, `stdout` и `stdprn` буферизуются системой. Когда буфер полон, он автоматически сбрасывает свое содержимое. Сброс выполняется и при завершении процедуры ввода (вывода). Буферизация обеспечивает передачу данных большими блоками, ускоряя процессы ввода-вывода.

Отметим, что при аномальном завершении выполнения программы содержимое буфера может остаться "непрочитанным", что приведет к потере информации.

С помощью функций `setbuf()` и `setvbuf()` можно лишить буфера или назначить буфер небуферизованному потоку. Буферы, назначаемые системой, пользователю недоступны. Однако буферы,

назначаемые посредством `setbuf()` или `setvbuf()`, определяются пользователем. Он может ими манипулировать как переменными. Буферы могут быть любых размеров. Если применяется функция `setbuf()`, размер задает константа `BUFSIZ`, описываемая в файле `stdio.h`. Если применяется функция `setvbuf()`, размер буфера задает пользователь.

В приводимой ниже программе открываются два файла `fl.txt` и `f2.txt`, а затем применяется функция `setbuf()`, позволяющая вместо автоматически назначаемого буфера размером `BUFSIZ` выбрать в качестве буфера переменную или область.

Пример:

```
#include <stdio.h>
char bufferf[BUFSIZ];
FILE *po1, *po2;
main()
{
    po1 = fopen("fl.txt","r");
    po2 = fopen("f2.txt","w");

    /* Для потока po2 назначается буфер пользователя */
    setbuf(po1,buffer);

    /* Для потока po2 отменяется буфер */
    setbuf(po2,NULL);
    printf("Манипуляции буферизации/дебуферизации закончены");
}
```

Функция `setvbuf` позволяет не только отменять буферизацию или назначить область для буфера, но и менять размер последнего. Эта функция используется, когда файл уже открыт как поток. В файле `stdio.h` она имеет прототип вида

`int setvbuf(FILE *поток, char *буфер,
int тип-буфера,тип-размера размер-буфера);`

Здесь "тип-буфера" задается определяемыми в `stdio.h` системными константами:

- `IOFBF` - полная буферизация;
- `IOLBF` - буферизация строки;
- `IONBF` - без буферизации.

Пример:

```
#include <stdio.h>
char buffer[1024];
FILE *po1, *po2;
int res;
main()
{
    po1 = fopen("fl.txt","r");
    po2 = fopen("f2.txt","w");
    if ( res = setvbuf(po1,buf,-IOFBF,
        sizeof(buffer)) != 0 )
```

```

printf("Ошибочен тип или размер буфера \n");
else
printf("Буфер потока po1 равен 1024 байт\n");
if ( setvbuf(po1,NULL,-IOFBF,0) != 0 )
printf("Ошибочен тип или размер буфера\n");
else
printf("Теперь поток po2 не имеет буфера\n");
}

```

Очистка (сброс) буфера выполняется автоматически по мере его заполнения. Пользователь может очистить буфер потока, подав команду (вызвав функцию) `fflush()`, прототип которой имеет следующий вид:

```
int fflush(FILE *поток);
```

Если поток открыт для записи (`write`), `fflush()` сбрасывает содержимое буфера в соответствующий файл ("пишет" в него). Если поток открыт для чтения (`read`), то очистка буфера приводит к потере данных. Например:

```
fflush(po1);
```

Для сброса всех буферов открытых потоков применяется функция `flushall()`. Ее прототип имеет вид:

```
int flushall(void);
```

Например:

```
flushall();
```

Эта функция возвращает в качестве значения количество открытых файлов (потоков).

Для закрытия одного или всех открытых потоков используется функция `fclose()` или `fcloseall()` соответственно. Однако эти функции не воздействуют на стандартные потоки `stdin`, `stdout`, `stderr`, `stdin`, `stdout`, `stderr`.

Если в программе пользователя явно файлы не закрываются, то это происходит автоматически по завершении работы программы. Рекомендуется неиспользуемые потоки закрывать явно, поскольку максимально допустимое число одновременно открытых файлов всегда ограничено.

Напомним, что информация о состоянии открытого потока хранится в структуре типа `FILE`, описываемой в файле `stdio.h`. В этой структуре хранится не только информация о позиции файла, указатель на соответствующий буфер, но также индикатор возникновения ошибки и индикатор конца файла. Если при работе с открытым потоком обнаруживается ошибка, то индикатору присваивается значение, отличное от нуля. Макрос `ferror()` проверяет, нет ли ошибки в операциях ввода-вывода указанного потока. Если ошибка ввода-вывода потока имеет место, индикатор ошибки остается "установленным", пока поток не будет закрыт, "перемотан" в начало (`rewind`) или не будет обработан функцией `clearerr()`.

В файле `stdio.h` макрос `ferror()` имеет прототип вида

```
int ferror(FILE *поток);
```

Если в потоке ошибка не обнаружена, значение функции `ferror()` равно нулю, иначе оно отлично от нуля.

Пример:

```
/* Программа, обнаруживающая ошибку (если она имеется), а затем  
"сбрасывающая" на нуль индикатор ошибки */
```

```
#include <stdio.h>
```

```
FILE *pointer;
```

```
char *stroka = "Язык Си";
```

```
main()
```

```
{  
    pointer = fopen("Test1.doc", "r");  
    fprintf(pointer, "%s \n", stroka);  
    if ( ferror(pointer) )  
    {  
        fprintf(stderr, "Ошибка: попытка писать в файл,  
        открытый для чтения \n"); clearerr(pointer);  
    }  
}
```

В файле `stdio.h` определена функция

```
clearerr(FILE *поток)
```

Она перемещает указатель потока в его начало. Ее действие эквивалентно функции

```
fseek(поток, 0L, SEEK-SET);
```

Однако `fseek()` не "очищает" (сбрасывает) индикаторы ошибки и конца файла. Кроме того `fseek()` возвращает значение (равное нулю, если перемещение указателя потока выполнено успешно).

Функция `rewind` не возвращает никакого значения ("связанного" с ее именем).

5.4. Функции и макросы

Библиотечные подпрограммы, собранные в библиотеку и вызываемые путем указания их имени со списком параметров, называют функциями. Текст функции хранится в библиотеке в форме объектного кода, т.е. он подвергнут компиляции.

Макрос предполагает наличие макроопределения на языке Си. Директива препроцессора `#define` описывает выражение, определяющее значение макроса. При компиляции текст этого выражения помещается на место макроса. Как и функция, макрос может иметь аргументы. Главное достоинство макросов - меньшее время выполнения операций. (Машинный код макроса встроен в программу.) Однако макрос увеличивает объем памяти, занимаемый программой. (Особенно когда один и тот же макрос используется несколько раз.) Функция обеспечивает компактность программы. Однако вызов функции требует дополнительных затрат времени. Иногда в языке Си одно и то же реализуется и макросом и функцией. Например, для чтения одного символа из

открытого потока можно использовать макрос `getc()` или функцию `fgetc()`.

Широко применяемому макросу чтения символа из текущего потока `getc()`, имеющему прототип

```
int getc(FILE *поток);
```

“соответствует” функция `ungetc()`, используемая для возврата обратно в поток символа, оказавшегося нежелательным.

Макрос `getc()` “читает” один символ как целое число без знака и принимает целочисленное значение. При этом указатель позиции в потоке (файле) перемещается в следующую позицию. Если достигается конец файла, то значение `getc()` становится равным EOF.

Функция `ungetc` определена в файле `stdio.h` посредством прототипа

```
int ungetc(int символ, FILE *поток);
```

Эта функция “возвращает” символ обратно в поток и очищает индикатор “конец файла”. Последующая операция чтения открытого потока начинается с “возвращенного символа”. Физически символ помещается в буфер.

Символы, помещаемые обратно в поток, можно “стереть” посредством функции `fflush()`, `fseek()`, `fsetpos()` или `rewind()`.

После успешного выполнения `ungetc()` для текстового потока указатель позиции в файле имеет “неопределенное” значение, пока не будут прочитаны все “буферизованные” символы или пока они не будут “стерты” (буфер очищен).

После успешного выполнения функции `ungetc()` для бинарного потока указатель позиции в файле получает отрицательное приращение. Функцию `ungetc()` можно применять, если прочитан хотя бы один символ.

Пример:

```
/* Программа преобразования символьного представления целого числа без знака в  
целое число. Если встречается символ, отличный от цифры, то выводится число и  
этот символ */
```

```
#include <stdio.h>
```

```
#include <ctype.h>
```

```
FILE *поток;
```

```
int ch;
```

```
int res = 0;
```

```
main()
```

```
{
```

```
    поток = stdin;
```

```
    printf("Введите число:");
```

```
    while ((ch=getc(поток))!=EOF && isdigit(ch))
```

```
        res = res * 10 + ch;
```

```
    if (ch != EOF)
```

```
        ungetc(ch,поток);
```

```
        printf("Число = %d\ Очередной символ = %c\n",
```

```
            res, getc(поток));
```

```
}
```

В файле `ctype.h` макрос `isdigit()` определен в виде

```
int isdigit(символ);
```

Этот макрос принимает нулевое значение, если “символ” - это цифра 0,1,2,...,8 или 9. В противном случае, значение макроса равно нулю.

Отметим, что для записи символа в поток применяются функция `fputc()` или макрос `putc()`. Для чтения из потока строки символов используется функция `fgets()`. Если роль потока играет `stdin`, то можно использовать функцию `gets()`. Прототип функции `fgets()` в файле `stdio.h` имеет следующий вид:

```
char *fgets(char *строка, int число—символов,  
FILE *поток);
```

Эта функция читает из входного потока очередные символы, включая признак начала новой строки (`\n`). Всего читается $(m-1)$ символ, где m - количество символов, задаваемое вторым аргументом функции. Результат записывается в виде строки (определяемой как массив символов первым аргументом функции). Всего читается $(m-1)$ символ потому, что к прочитанным символам добавляется признак конца строки символов (символ `\0`). Если $m = 1$, то строка имеет вид: `““`. Отметим, что функция `fgets()` подобна `gets()`. Однако, если функция `fgets()` “встречает” код признака новой строки (символ `\n`), то `fgets` его “читает”. Функция `gets()` заменяет `\n` на `NULL`.

При успешном выполнении `fgets()` “возвращает” строку символов. При неудаче эта функция возвращает признак конца файла или ошибки. Используя `feof()` или `ferror()`, можно проверить результат.

Пример:

```
/* Воспроизведение на экране строки текста из файла */  
#include <stdio.h>  
FILE *file-point;  
char stroka[80], *resu-point;  
main()  
{  
    file-point = fopen("test1.c", "r");  
    resu-point = fgets(stroka, 80, file-point);  
    printf("%s", stroka);  
    fclose(file-point);  
}
```

5.5. Функции, оперирующие с файлами на “нижнем” уровне

Эти функции не выполняют ни буферизацию, ни форматирование данных. Здесь файлы адресуемы посредством так

называемой ссылки на файл (file handle). Это целое число, используемое операционной системой для обращения к файлу. Набор соответствующих функций содержит свои прототипы в файле `io.h`.

Имена наиболее часто используемых функций: `open`, `read`, `write`, `lseek`, `tell`, `close`, `creat`, `dup` и т.д.

В отличие от функций, манипулирующих файлом как потоком, функции "нижнего уровня" не нуждаются в файле `stdio.h`. Исключение составляют ситуации, когда требуются объекты, определенные в `stdio.h` (например, константа `EOF` и т.п.).

Открытие файла

Перед выполнением операций ввода-вывода файл должен быть открыт посредством функции `open()`, `sopen()` или `creat()`. Файл можно открыть для записи, чтения, а также для того и другого одновременно. Файл можно открыть как текстовый или двоичный. Для указания "цели" открытия файла (запись в конец файла, открытие как двоичного или текстового файла, открытие только для чтения и т.п.).

Функция `open` использует "флаги", описываемые в файле `fcntl.h`. Поэтому директивой включения его присоединяют к программе. В некоторых случаях для открытия файла требуется присоединение файлов `sys/stat.h` и `sys/types.h`. Указанные функции открытия файла возвращают в качестве значения ссылку на файл (file handle).

Пример (фрагмент программы):

```
...
int handle-1, handle-2 ;
...
handle-1 = open("file1.txt", O_RDONLY);
if ( handle-1 == -1 )
    printf("Файл открыть не удалось.");
else
    printf("Файл открыт только для чтения.");
handle-2 = open("file2.txt", O_APPEND);
printf("Попытка открыть файл
      для добавления записей");
```

Ссылка на файл имеет целочисленное значение (положительное при благополучном исходе). В последующих операциях с открытым файлом ссылка на файл используется как средство адресации к нему.

Ссылки на файлы, назначаемые системой, так называемым стандартным файлом ввода, вывода, приема сообщений об ошибках, внешнего устройства и принтера присваиваются (системой по умолчанию) ссылки 0, 1, 2, 3 и 4 соответственно. Если эти файлы открываются как потоки, им присваиваются имена `stdin`, `stdout`, `stderr`, `stdaux` и `stdprn`. Говоря иначе, между

стандартными потоками и стандартными ссылками на файлы имеем следующее соответствие:

stdin	0
stdout	1
stderr	2
stdaux	3
stdprn	4

“Стандартные” файлы открываются автоматически в начале работы программы. Ссылками на эти файлы можно пользоваться сразу же (без каких-либо предварительных операций). Для получения значения ссылки файл можно использовать функцию `fileno()`. Так приведенное выше соответствие между именами стандартных потоков и значениями их ссылок можно получить, выполнив программу:

```
#include <stdio.h>
...
main()
{
    printf("stdin: %d\n", fileno(stdin));
    ...
    printf("stdprn: %d\n", fileno(stdprn));
}
```

Результат выполнения программы:

stdin: 0

...
stdprn: 4

Чтение данных из файла и в файл

Чтение содержимого файла и запись данных в него выполняются с двух основных функций `read()` и `write()`. Файл `io.h` содержит описание прототипа функции чтения в виде

```
int read(int ссылка, char * буфер,
         unsigned int количество);
```

Эта функция пытается читать указанное количество байтов в заданный буфер из файла, адресуемого ссылкой (`file handle`). С именем функции связано возвращаемое ею значение; в данном случае это количество фактически прочитанных байтов. Если возвращаемое значение равно нулю, то выполняется попытка читать из файла, “находясь” у его конца. Если возвращаемое значение равно - 1, имеет место ошибка чтения. При этом специальный параметр `errno` получает значение `EBADF`, что означает некорректность используемой ссылки на файл. При

благополучном чтении количество прочитанных байтов для текстового режима может оказаться меньше, чем для двоичного. Известно, что если файл воспринимается как текстовый, каждая пара символов CR (возврат каретки) и LF (перевод строки) интерпретируется как один символ CR.

Пример:

```
/* Программа чтения из файла в буфер 40 тыс. байт */
```

```
#include <fcntl.h>
/* В файле fcntl.h описан флаг O_RDONLY */
#include <io.h>
#include <stdio.h>
char buffer[40000];
main()
{
    int f-handle;
    unsigned int koli = 40000, bytre;

    /* Открытие файла только для чтения только с проверкой корректности этой
    операции */
    if ((f-handle=open("file1.txt",O_RDONLY))==-1)
    {
        perror("Неудача !");
        exit(1);
    };

    /* Чтение из файла в буфер */
    if ((bytre=read(f-handle,buffer,koli))<=0)
        perror("Файл не читается !");
    else
        printf ("Прочитано %u байтов из файла \n",
koli);
}
```

Здесь функция `perror()` помещает в стандартный поток `stderr` сообщение об ошибке. Прототип функции `perror()` описан в файле `stdio.h` и имеет следующий вид:

```
void perror(const char *строка);
```

На стандартное устройство приема сообщений об ошибках `stderr` выводится "строка", затем ставится запятая и после нее помещается системное сообщение об ошибке. Номер ошибки регистрируется в переменной `errno`, описываемой на внешнем уровне.

Функция `exit()` очищает все буферы и закрывает все файлы перед завершением процесса. Обобщенная форма этой функции:

```
exit(статус)
```

Если "статус" задан в виде нуля, имеет место "нормальный" выход из процесса. Если "статус" не равен нулю, то другой процесс, ожидающий исхода текущего процесса, получает информацию об аномальном завершении. "Статус" доступен команде `IF_ERRORLEVEL` так называемого пакетного (batch) файла MS-DOS (например, файла `AUTOEXEC.BAT`).

Файл `io.h` содержит прототип функции записи в файл в следующем виде:

```
int write(int ссылка, char *буфер,  
          unsigned int количество);
```

Запись в файл начинается с текущей позиции указателя файла. С именем функции `write()` связан целочисленный результат - количество записанных в файл байтов. Если возвращаемое функцией значение равно `-1`, имеет место ошибка записи. При этом переменная `errno` получает одно из двух значений: `EBADF` - неверная ссылка на файл или файл не открыт для записи; `ENOSPC` - не осталось места для размещения данных.

Итак, функции нижнего уровня манипуляций с файлами сигнализируют о возникновении ошибки присваиванием соответствующего значения переменной `errno`. Используя функцию `feof()`, можно вывести информацию об ошибке ввода-вывода (с сообщением об ошибке). В процессе чтения файла или записи в него указатель позиции в файле (сокращенно "указатель файла") перемещается по файлу. Для проверки "достигнут ли конец файла" можно использовать функцию `eof()`. Ее описание дано в файле `io.h` и имеет следующий вид:

```
int eof(int ссылка);
```

где "ссылка" - это ссылка на файл (`file handle`). Если достигнут конец файла, значение функции равно `1`; иначе оно равно `0`. Если значение этой функции равно `-1`, то имеет место ошибка. При этом переменной `errno` присваивается значение `EBADF`.

Для позиционирования указателя файла используется функция `lseek()`. Заметим, что указатель можно "поставить" после конца файла, но не перед ним.

Прототип функции `lseek()` имеет следующий вид:

```
long lseek(int ссылка, long смещение,  
           int начало-отсчета);
```

"Начало-отсчета" должно быть одной из констант, определенных в файле `stdio.h`, а именно:

`SEEK-SET` - начало файла;

`SEEK-CUR` - текущая позиция в файле;

`SEEK-END` - конец файла.

Параметр "смещение" - это количество байтов, указывающих положение текущей позиции в файле относительно начала отсчета. С именем функции `lseek()` связано возвращаемое ею значение - смещение текущей позиции относительно начала файла. Если это значение равно `-1`, то имеет место ошибка; переменная `errno` получает значение `EBADF` - неверная ссылка на файл. Другим возможным значением `errno` может быть `EINVAL` - неверное начало отсчета или смещение за начало файла.

Пример:

...

```

file-point = open("file1.txt", O_RDONLY);
...
pos = lseek(file-point, 12L, SEEK-CUR);
if (pos == -1L)
    perror("Текущая позиция не найдена.");
else
    printf("Текущая позиция = %d \n", pos);
...

```

Для определения текущей позиции в файле в байтах относительно начала файла используется функция `tell()`. Она описана в файле `io.h` и имеет следующий вид:

```
long tell(int ссылка);
```

Если имеет место ошибка, то эта функция возвращает `-1L`.

Для закрытия файла служит функция `close()`. Ее описание в файле `io.h` представлено в следующем виде:

```
int close(int ссылка);
```

В случае благополучного исхода выполнения функции ее значение равно 0, иначе `-1`.

5.6. Функции, оперирующие с консолью и портами ввода - вывода

Прототипы этих функций описаны в файле `conio.h`. Эти функции можно рассматривать как дополнение к функциям, оперирующим с потоками. Они позволяют "читать" с консоли и "писать" на нее. (Роль консоли выполняют клавиатура и дисплей.) Они также позволяют читать и писать данные через порт (такой, например, как порт принтера). Функции ввода - вывода через порт просто манипулируют байтами. Аналогичные функции работы с консолью имеют некоторые дополнительные возможности. Например, можно фиксировать (устанавливать) факт нажатия клавиши. Можно вводить с клавиатуры символы без отображения на экране дисплея и т.п.

К функциям ввода с консоли относятся: `cgets`, `cscanf`, `getch`, `getche`, `kbhit`. Функции записи на консоль: `cprintf`, `cputs`, `putch`, `ungetch`. Объект, на который эти функции вводят или выводят данные, может быть переопределен на уровне операционной системы.

Консоль или порт не требуют "открытия" перед использованием подобных функций и "закрытия" по окончании работы. Для чтения или записи через порт одного байта одновременно используются функции `inpr()` и `outpr()` соответственно. Для подобных операций одновременно с двухбайтовым словом применяются функции `inprw()` и `outprw()` соответственно. Так называемые "консольные" функции ввода-вывода делятся на оперирующие строками символов (`cgets` и `cputs`), форматированными данными (`cscanf` и `cprintf`) и отдельными символами (`putch`, `getch`, `getche`, `ungetch`, `kbhit`).

Функция `putch()` “пишет” символ на консоль, функции `getch()` и `getche()` “читают” символ с консоли; при этом `getche()` “выдает эхо” читаемого символа на консоль, а `getch()` “эхо” не выдает; функция `kbhit()` устанавливает факт нажатия клавиши. Если клавишу нажать, то `kbhit()` возвращает целочисленное значение, отличное от нуля; иначе значение функции равно нулю.

Пример:

```
/* Программа “ждет” нажатия любой клавиши */
#include <conio.h>
#include <stdio.h>
main()
{
    printf("Ожидаем нажатия любой клавиши \n");
    while ( !kbhit() );
    printf("Была нажата клавиша: %c \n", getch());
}
```

Цикл ожидания длится до тех пор, пока не будет нажата любая клавиша. Значение функции `kbhit()` помещается в буфер. Эта функция (а также и `getche()`) “очищает” буфер.

Функции `getch()` и `getche()` “читают” символ с консоли и возвращают символ в виде целого числа. Возможная ошибка не фиксируется. При чтении функциональной клавиши или клавиши курсора функцию нужно вызвать дважды.

Пример:

```
/* Чтение символов с консоли и отображение первого символа, отличного от пробела */
#include <conio.h>
#include <ctype.h>
int s;
main()
{
    printf("Вводите символы: \n");
    do
    {
        s = getch();
        while ( isspace(s) );
        putch(s);
    }
}
```

Функция `ungetch` помещает очередной “прочтенный” символ в буфер, делая его следующим читаемым символом.

Ниже приведен текст программы, выполняющей следующее: читается слово (строка символов) длиной не более 49 символов. Слово “обрывается” очередным пробелом. Помещенное в буфер слово дополняется символом “\0” в качестве признака конца строки символов. Функция `ungetch()` позволяет в данном случае поместить исходный разделитель слов (пробел) в буфер для последующего возможного использования. (Любое последующее чтение иницирует извлечение из буфера.)

Пример:

```
#include <conio.h>
#include <ctype.h>
```

```

#include <stdio.h>
char buffer[50];
int count = 0;
int sim;
main()
{
    sim = getche();
    /* Цикл, пропускающий предшествующие пробелы */
    while ( isspace(sim) )
        sim = getche();
    /* Цикл чтения слова */
    while ( count < 49 )
    {
        if (isspace(sim)) break;
        buffer[count++] = sim;
        sim = getche();
    };
    ungetch (sim);
    buffer[count] = '\0';
    printf("\n Строка символов: %s \n", buffer);
}

```

6. СРЕДА MS-DOS: ПЕРЕДАЧА ПАРАМЕТРОВ ФУНКЦИИ MAIN И ВЫЗОВ ПОДЧИНЕННЫХ ПРОЦЕССОВ

6.1. Передача параметров функции main

Си-программа - это совокупность функций, одна из которых - main(). С выполнения main() начинается и, как правило, в ней завершается работа программы. При вызове любой функции можно организовать передачу параметров.

При старте программы с командной строки после имени программы можно указать один или несколько разделяемых пробелами параметров. Они будут восприняты функцией main как перечень фактических параметров.

Согласно синтаксису языка Си функция main(), которой посылаются параметры (фактические), "ожидает" их приема, располагая двумя формальными параметрами. Их принято именовать argc и argv (хотя с равным успехом допустимо использование других наименований).

argc - целочисленный параметр, указывающий число фактических параметров, указываемых после имени программы в командной строке. Например:

prog1 Система Turbo-C

Второй формальный параметр argv - это ссылка (указатель) на массив символьных строк; каждая строка символов, а точнее массив символов, - это место для размещения одного из фактических параметров, указываемых в командной строке после имени программы.

По соглашению, принятому в системах программирования Turbo-C и MicroSoft-C, имя программы занимает первый по порядку, нулевой по номеру, элемент массива `argv[]`, а именно `argv[0]`. В нашем примере в элементе `argv[1]` разместится слово "Система", а в элементе `argv[2]` - "Turbo-C".

Положим, что программа `prog1.c` предназначена для отображения (выдачи эха) на экране текста параметров вводимой с клавиатуры командной строки. Например:

`prog1 Система Turbo-C`

или

`prog1 Хочу знать Си`

Возможный вариант текста программы `prog1.c`:

```
#include <stdio.h>
main (argc,argv)
int argc;
{
    int k;
    for (k=1; k<argc; k++)
        printf("%s%c",argv[k],(k<argc-1)?` `:`\n`));
}
```

Переменная `argv` - ссылка на массив ссылок. Говоря иначе, команда

`*argv[];`

эквивалентна команде

`char **argv;`

В результате возможен следующий вариант программы `prog1.c`:

```
#include <stdio.h>
main (argc,argv)
int argc;
char **argv;
{
    while (--argc>0)
        printf ("%S%C",*++argv,argc>1?` `:`\n`);
}
```

Расширением стандарта языка Си (ANSI C) является возможность использования третьего формального параметра функции `main`, именуемого обычно `envp`. Если `argv` можно назвать ссылкой на вектор параметров-строк, то `envp` - ссылка на вектор переменных среды.

Каждая переменная среды - это строка символов вида

имя-перем-среды = значение

Переменные среды ("оболочки") могут использоваться для

настройки параметров среды, в которой программа выполняется. (В частности это позволяет, например, настроить среду операционной системы XENIX для выполнения Си - программы, разработанной для среды MS-DOS.)

Пример программы, воспроизводящей имя программы и вводимые с командной строки параметры, а также переменные среды:

```
#include <stdio.h>
main (int argc, char **argv, char **envp)
{
    int k;
    for (k=0; k<argc; k++)
        printf("argv[%d]=%s\n", k, argv[k]);
    while(*envp != 0)
        printf("%s\n", *envp++);
}
```

6.2. Среда MS-DOS

Начиная с MS-DOS версии 2.0 существовала среда (environment). Это специальная область оперативной памяти, именуемая МЕВ (Master Environment Block). Dos выделяет МЕВ для хранения имен переменных "среды" (и их значений). Блок памяти, именуемый МЕВ, легко доступен пользователю. Его содержимое пользователь может модифицировать прямо из командной строки или запуском батч-файла (*.bat).

Переменная среды может быть помещена в МЕВ командой
SET имя = значение

Чтобы воспроизвести на экране имена и значения всех переменных среды достаточно подать команду
SET

Некоторые имена являются зарезервированными именами переменных среды DOS, например: PATH, PROMPT, COMSPEC. Значением COMSPEC является наименование используемого командного процессора. DOS создает COMSPEC и PATH по умолчанию, если это не делает программист явно в файле AUTOEXEC.BAT.

Поскольку PROMPT, PATH и COMSPEC - специально зарезервированные слова, им можно присваивать значения даже без указания служебного слова SET.

PROMPT = Master!
PATH = C:\;C:\DOS

Эти команды можно подавать с командной строки или поместить в батч-файл.

Одиночной переменной среды можно присвоить несколько значений, разделяя их точкой с запятой. Например, в файл AUTOEXEC.BAT можно включить команду

SET USERS = IVANOV;PETROV;SEDOV

Для доступа к переменной среды в батч-файле нужно слева и

справа от ее имени поместить знак процента (%имя%).

Положим, что переменная **USERS** используется для санкционированного доступа Иванова, Петрова, Седова к работе на ПЭВМ в среде, "уточняемой" файлом **UNLOCK.BAT**. Последний реализует защиту от несанкционированного доступа. Файл **UNLOCK.BAT** может включать следующие команды:

```
:CHECK
FOR %%a IN (%USERS%) DO
IF "%1"=="%%a" GOTO A
:INVALID
ECHO =====
ECHO = Попытка несанкц.доступа
ECHO =====
GOTO END
:A
ECHO Здравствуйте,%1
ECHO Загружается программа...
cd c:\secret
TC
:END
```

Пояснение: пользователь должен набрать на клавиатуре, например,

UNLOCK PETROV

Возможности блока **MEB** ("среды") нельзя использовать неограничено. Этот блок формируется в момент загрузки операционной системы. Далее действует правило: каждая программа или процесс, выполняемые в данный момент, получает копию "среды", которой пользуется программа, запустившая текущую программу (процесс). Например, **DOS** загружает **COMMAND.COM** для обработки наших **DOS**-команд. При этом **COMMAND.COM** получает копию **MEB**. Мы пускаем на выполнение какую-либо программу, а затем возвращаемся к услугам **COMMAND.COM** с указанной копией. Например, **Turbo-Basic** и некоторые текстовые процессоры из своей среды позволяют выполнять команды **DOS** (это достигается загрузкой второго **COMMAND.COM**, т.е. копии командного процессора). Естественно появляется дубль "среды", т.е. блока **MEB**, поскольку "новый" **COMMAND.COM** получает копию "среды").

Если в "среду" вносятся некоторые изменения, то по выходе из копии **COMMAND.COM** теряется копия среды и все изменения. Все это определяет круг проблем и ограничений на размер блока **MEB**, т.е. "среды". Заметим, что **DOS** имеет команду

COMMAND /C команда-DOS

где параметр "**/C**" обеспечивает возврат к исходному **COMMAND.COM** после того, как вызвана копия для выполнения "команды-DOS".

Отметим, что каждый вызов копии COMMAND.COM требует дополнительно около 3000 байт.

О размере "среды"

Блок MEB для DOS 2.0 составляет 160 байт. (Не менее 29 байт берут переменные PATH и COMSPEC, оставляя пользователю 131 байт.) DOS 3.1 и "выше" допускает "регулировку" размера среды посредством команды SHELL в файле CONFIG.SYS. Формат команды:

SHELL = COMMAND.COM /P /E:размер

Ключ (параметр) "/P" "приказывает" DOS загружать AUTOEXEC.BAT автоматически. Без этого ключа данный файл не загружается. Ключ "/E" позволяет пользователю задать размер "среды". Для DOS 3.2 и "выше" этот размер можно установить, выбрав число байтов из диапазона 160 - 32767. Эта версия DOS (и выше) позволяют изменить размер "новой среды". Для этого команду COMMAND... следует дополнить ключом "/E:размер", где "размер" выбирается из диапазона значений 127 - 32767 байт. Например,

COMMAND /E:2000

создает среду размером 2000 байтов.

6.3. Вызов подчиненного процесса из текущего

В среде MS - DOS файл типа EXE или COM является выполнимым, то есть вызываемым из командной строки или из текущей выполняемой программы (текущего процесса). Применяя термин "процесс" к пригодному к выполнению файлу, можно выделить две группы библиотечных функций - EXEC... и SPAWN... - для вызова одного процесса из другого. Вызываемый процесс можно назвать подчиненным. Для использования этих функций нужно подключить к тексту вызывающей программы (процессу) файл process.h.

Функции типа EXEC

В файле process.h определены следующие параметры прототипов функций типа EXEC:

```
int execl (имя, arg0, ..., argN, NULL);
int execlp (имя, arg0, ..., argN, NULL);
int execlpe (имя, arg0, ..., argN, NULL, envp);
int execv (имя, argv);
int execve (имя, argv, envp);
```

```
int execvp (имя, argv);  
int execlpe (имя, argv, envp);  
char *имя;  
char *arg0, *arg1, ..., *argN;  
char *argv[];  
char *envp[];
```

где "имя" - это имя файла (процесса), подлежащего выполнению (перед именем следует поместить префикс - маршрут доступа к файлу);

arg0, ..., argN - список параметров (аргументов), передаваемых вызываемому процессу; первым по порядку (нулевым по номеру) параметром указывается ссылка на имя вызываемого файла (как принято соглашениями в системе C).

argv[] - массив указателей на параметры (строки символов);

envp[] - массив указателей (ссылок) на переменные среды.

Каждая из этих функций загружает в память и иницирует выполнение программы (процесса) с указанным именем. Если функция выполнена успешно, вызываемый процесс размещается на место положения вызывающего процесса. При неудаче значение функции EXEC... равно -1. При этом системная переменная errno принимает одно из следующих значений:

E2BIG - список аргументов или область памяти для переменных среды за пределами допустимого диапазона значений (128 байт - предельная длина списка параметров, 32K - предельный объем памяти, предоставляемый переменным среды);

EACCESS - выполнение вызываемого процесса (файла, используемого в многопользовательском режиме) приводит к нарушению правила его совместного использования или защиты от несанкционированного доступа;

EMFILE - ситуация, когда предпринимается попытка открыть слишком много файлов (чтобы "понять", является ли вызываемый файл выполнимым, его прежде всего нужно открыть);

ENOENT - не найден файл с указанным именем или маршрутом (ошибочен маршрут или имя файла);

ENOEXEC - вызываемый файл не является выполняемым;

ENOMEM - недостаточно памяти для размещения вызываемого процесса (файла) или имеют место дефекты в области, выделенной для размещения.

Все функции группы EXEC помимо общей части имени (EXEC) имеют специфические компоненты (буквы) в своем наименовании. Эти буквы означают следующее:

r - используется переменная среды PATH для поиска файла, подлежащего выполнению;

l - список передаваемых параметров должен состояться как последовательность отдельных имен;

v - вызываемому процессу список параметров (аргументов)

передается в виде массива ссылок (argv[]) на параметры (строки символов);

e - передача вызываемому процессу массива ссылок (envp[]) на переменные среды.

Замечание: при формировании массива ссылок argv[] на параметры элемент argv(0) должен быть указателем на параметр в форме строки символов - имени вызываемого файла, а argv[n+1] должен иметь значение null, т.е. быть ссылкой (указателем) в "никуда".

Пример: данная программа вызывает файл CHD.EXE, который занимает ее место.

```
#include <stdio.h>
#include <process.h>
main()
{
    int st;
    st = execl("CHD.EXE","CHD.EXE",
              "Парам1","Парам2",NULL);
    printf ("Неудачное выполнение EXECL.
            Код равен %d\n",st);
}
```

Пояснение: выполнение printf() возможно только при неуспехе вызова подчиненного процесса CHD.EXE.

Текст (исходный) вызываемого файла - программа CHD.C:

```
#include <stdio.h>
main (int argc, char *argv[])
{
    int k;
    for (k=0; k<argc; k++)
        printf("argv[%d] = %s \n", k, argv[k]);
}
```

Ожидаемый результат работы этой программы:

```
argv[0] = CHD.EXE
argv[1] = Парам1
argv[2] = Парам2
```

Функции типа SPAWN

Прототипы функций типа SPAWN используют следующие параметры:

```
int spawnl(режим,имя,арг0,...,аргN,NULL);
int spawnle(режим,имя,арг0,...,аргN,NULL,envp);
int spawnlp(режим,имя,арг0,...,аргN,NULL);
int spawnlpe(режим,имя,арг0,...,аргN,NULL,envp);
```

```

int spawnv(режим,имя,argv);
int spawnve(режим,имя,argv,envp);
int spawnvp(режим,имя,argv);
int spawnvpe(режим,имя,argv,envp);
int фп;
char *имя;
char *apг0,*apг1,...*apгN;
char *argv[ ];
char *envp[ ];

```

Параметр “режим” (точнее, флаг режима) принимает одно из следующих значений: P-WAIT (вызывающий процесс ждет завершение вызываемого), P-NOWAIT (вызываемый и вызывающий процесс протекают одновременно; режим не реализован), P-OVERLAY (вызываемый процесс размещается на месте вызывающего, уничтожая его).

Как и в случае функций типа EXEC данная группа функций помимо общей части имени (а именно, SPAWN) имеет в составе имени одну или несколько из следующих букв: p, l, v, e. Их назначение и правила определения аргументов функций SPAWN такие же, как функций EXEC.

Если функция типа SPAWN терпит неудачу при выполнении, системная переменная еггп получает одно из следующих значений: E2BIG, EINVAL, ENOENT, ENOEXEC, ENOMEM. “Смысл” этих значений такой: же, как и в случае “неуспеха” функции EXEC. Значение EINVAL “говорит” о том, что недопустимым выбрано значение флага режима (modeflag). Если значение флага режима является P-WAIT и функция типа SPASWN успешно выполнена, то эта функция “возвращает” в качестве своего значения так называемый “статус выхода” (exit status). Параметр “статус выхода” задается в качестве аргумента функции exit, имеющей прототип void exit (статус). В бэтч-файле MS-DOS значение статуса доступно команде IF-ERRORLEVEL параметр... Она сравнивает “статус” со своим “параметром” и позволяет по их совпадению выполнять действие, указываемое программистом.

Пример (текст вызывающего файла SP.C):

```

#include <stdio.h>
#include <process.h>
main()
{
    int st;
    st = spawnl (P-WAIT, "SUBP.EXE", "SUBP.EXE",
                "Параметр1", "Параметр2", NULL);
    if (st == -1)
        printf("Ошибка при вызове
                подчиненного процесса\n");
    else

```

```

printf ("Теперь управление
        снова передано SP.C.
        Код возврата равен %d\n",st);
exit(0);
}
/* Текст вызываемого процесса - файла SUBP.C */
#include <stdio.h>
main (int argc, char *argv[])
{
    int k;
    for (k=0;k<argc;k++)
        printf ("argv[%d]=%S\n",k,argv[k]);
    exit (100);
}

```

Оглавление

1.	<i>Базовые средства языка</i>	3
1.1.	Вывод данных	3
1.2.	Строки символов	4
1.3.	Операции над данными	6
1.4.	Операции над адресами	7
1.5.	Ввод данных	8
1.6.	Команды передачи управления	9
2.	<i>Функции</i>	13
2.1.	Определение функций	15
2.2.	Описание функций	16
3.	<i>Расширение базового подмножества средств языка программирования</i>	17
3.1.	Указатели	17
3.2.	Динамическое выделение памяти	18
3.3.	Указатели и функции	19
3.4.	Операции с указателями	20
3.5.	Массивы	20
3.6.	Массивы и указатели	21
3.7.	Массивы и строки	22
3.8.	Многомерные массивы	22
3.9.	Массивы и функции	23
3.10.	Структуры	24
3.11.	Структуры и указатели	25
3.12.	Передача управления: оператор switch	25
3.13.	Передача управления: дополнительные средства ее организации	27
3.14.	Классы памяти и инициализация переменных	30
4.	<i>Препроцессор и его директивы в программе</i>	39
4.1.	Замена идентификаторов	40
4.2.	Макроопределения	40
4.3.	Включение файлов	40
4.4.	Компиляция по условию	42
5.	<i>Файлы данных и функции ввода-вывода</i>	43
5.1.	Функции, оперирующие с потоком	44
5.2.	Потоки, открываемые системой по умолчанию	47
5.3.	Буферизация файла, открываемого как поток	50
5.4.	Функции и макросы	53
5.5.	Функции, оперирующие с файлами на "нижнем" уровне	55
5.6.	Функции, оперирующие с консолью и портами ввода-вывода	60
6.	<i>Среда MS-DOS: передача параметров функции main и подчиненных процессов</i>	62
6.1.	Передача параметров функции main	62
6.2.	Среда MS-DOS	64
6.3.	Вызов подчиненного процесса из текущего	66

Научное издание

Виталий Константинович Потоцкий

РАБОТАЕМ НА ЯЗЫКЕ СИ

Редактор М. Ю. Григоренко

Художник Л. А. Степанов

Подписано к печати 26.02.92. Формат 60 × 84¹/₁₆.

Бумага офсетная. Печать офсетная. Усл. п. л. 4,5.

Тираж 20 000 экз. Заказ 62

МП «Малип», 129626, Москва, ул. Инициативная, д. 2

При участии А/О «Таймер»

Отпечатано в типографии ИПО «Полигран»

125438, Москва, Пакгаузное шоссе, д. 1

14-

Настольная книга пользователя ПК

Язык программирования Си — наиболее популярное средство создания как системных, так и прикладных программ на всех современных вычислительных машинах от микро- и мини- до суперЭВМ.

Си — это действительно язык номер 1 в современном мире вычислительной техники. Элегантный по форме, мощный и гибкий по своим возможностям — это язык не только для профессионалов, но и многочисленных пользователей ПЭВМ, для тех, кто стремится достичь совершенства в применении всего разнообразия средств современной вычислительной техники.

Работаем на языке Си